Introduction à la Programmation Orientée Objet

... et son application au C++

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Prérequis et organisation du cours

- Prérequis : bonne connaissance en programmation C et en algorithmique
- Organisation :
  - 9h de cours $\rightsquigarrow$ 6 séances de 1.5h (le lundi de 9h30 à 11h00)
  - 18h de TD $\rightsquigarrow$ 6 séances de 3h (le vendredi de 9h30 à 12h30)
- Évaluation :
  - 25% $\rightsquigarrow$ projet de programmation (seul ou en binôme)
  - 75% $\rightsquigarrow$ examen final

Rappels élémentaires de langage C

⇝ la fonction principale `main`

```
// hello.c

// Inclusion des fichiers d'entetes
#include <stdio.h>   // gestion des entrees/sorties
#include <math.h>    // utilisation des fonctions mathematiques

// Definition de la fonction principale
int
main( void )
{
  printf("Hello world !\n");   // Affichage de "Hello world !"
  return 0;                    // par convention, "return 0" indique qu'aucune
                               // erreur n'est survenue
}
```

```
$> gcc -Wall -o hello hello.c

$> ./hello
Hello world !
```

- Remarques :
  - la fonction `main` est le point d'entrée du programme ⇝ elle est indispensable
  - les instructions sont exécutées dans l'ordre d'apparition dans le programme

Rappels élémentaires de langage C

⇝ déclaration d'une variable

```
// rappel-variable.c

#include <stdio.h>

int
main( void )
{
  int x1;             // Declaration de l'entier x1
  int x2 = 18;        // Declaration et initialisation de l'entier x2
  x1 = 17;            // Affectation de 17 a l'entier x

  printf("Ma valeur entiere x1 vaut : %d.\n",x1);
  printf("Ma valeur entiere x2 vaut : %d.\n",x2);

  return 0;
}
```

```
$> gcc -Wall -o variables variables.c

$> ./rappel-variable
Ma valeur entiere x1 vaut : 17.
Ma valeur entiere x2 vaut : 18.
```

Rappels élémentaires de langage C

⇝ fonction et passage de paramètres

```c
// fonctions.c

#include <stdio.h>

int
addition_v1(int a, int b)           // c = a + b
{ int c; c = a + b; return c; }

void
addition_v2(int a, int b, int c)    // c = a + b
{ c = a + b; }

void
addition_v3(int a, int b, int *c)   // c = a + b
{ *c = a + b; }

int
main( void )
{
  int x1 = 17, x2 = 18;
  int c_v1 =    addition_v1(x1,x2);          // x1 et x2 sont passes par valeurs
  int c_v2 = 0; addition_v2(x1,x2,c_v2);     // c_v2 est passe par valeur
  int c_v3;     addition_v3(x1,x2,&c_v3);    // c_v3 est passe par adresse

  printf("Finalement c_v1 = %d, c_v2 = %d et c_v3 = %d.\n",c_v1,c_v2,c_v3);

  return 0;
}
```

Rappels élémentaires de langage C

⇝ fonction et passage de paramètres

```
// fonctions.c

// ...
int
main( void )
{
  int x1 = 17, x2 = 18;
  int c_v1 =    addition_v1(x1,x2);          // x1 et x2 sont passes par valeurs
  int c_v2 = 0; addition_v2(x1,x2,c_v2);     // c_v2 est passe par valeur
  int c_v3;     addition_v3(x1,x2,&c_v3);    // c_v3 est passe par adresse

  printf("Finalement c_v1 = %d, c_v2 = %d et c_v3 = %d.\n",c_v1,c_v2,c_v3);

  return 0;
}
```

```
$> gcc -Wall -o fonctions fonctions.c

$> ./fonctions
Finalement c_v1 = 35, c_v2 = 0 et c_v3 = 35.
```

- **Remarque** :
  - une variable passée par valeur, n'est pas modifiée à l'extérieure de la fonction
    ⇝ utilisation du passage par adresse (`&c_v3`)

Rappels élémentaires de langage C

⇝ définition et utilisation de structures

```c
// structures.c

#include <stdio.h>

struct node
{
  int key;
  struct node* f_g;
  struct node* f_r;
};
typedef struct node node_t;


int
main( void )
{
  node_t n1;                                          // ou : struct node n1
  node_t* n2 = (node_t*)malloc(sizeof(node_t)); //       struct node* n2

  // Utilisation: n1.f_g et n1.f_r
  //              n2->f_g et n2->f_r

  free(n2);          // Liberation de la memoire allouee par malloc

  return 0;
}
```

Et maintenant la Programmation Orientée Objet et le C++

Pour apprendre le langage C++, le point essentiel consiste à se concentrer sur les concepts et à éviter de se perdre dans les détails techniques.
B. Stroustrup, *Le langage C++ - Edition spéciale* - §1.1.2 (2000)

Plan du cours

1. Pourquoi la programmation orientée objet ?
2. Principes fondamentaux de la programmation orientée objet
3. Éléments de modélisation d'un programme orienté objet
4. Quelques éléments importants du C++
5. Premier exemple de programme C++

Plan du cours

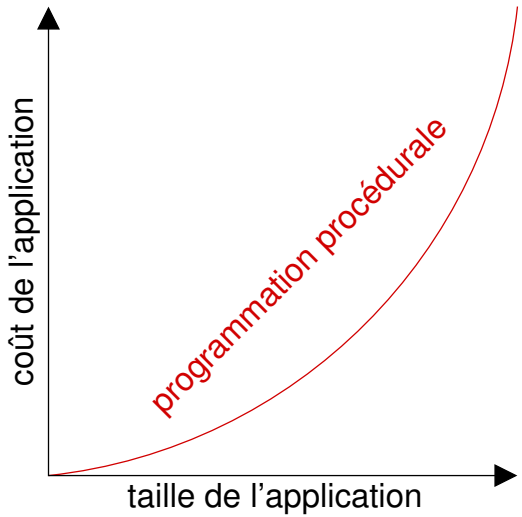
Qu'est ce que la programmation orientée objet ?

- Programmation Orientée Objet (POO) : paradigme de programmation informatique
  - élaboré par Alan Kay, dans les années 70'
  - définition et interactions de briques logicielles $\rightsquigarrow$ objets
- Un objet = un concept, une idée ou une entité du monde physique
  - par exemple : une voiture, un étudiant, ...
  - possède une structure interne et un comportement
- Quelques langages objets : C++, Java, Ada, PHP, Python, ...

Pourquoi la programmation orientée objet ?

- Depuis plusieurs années :
  - le matériel de moins en moins cher, et de plus en plus puissant
  - le développement d'applications de plus en plus performantes et complexes

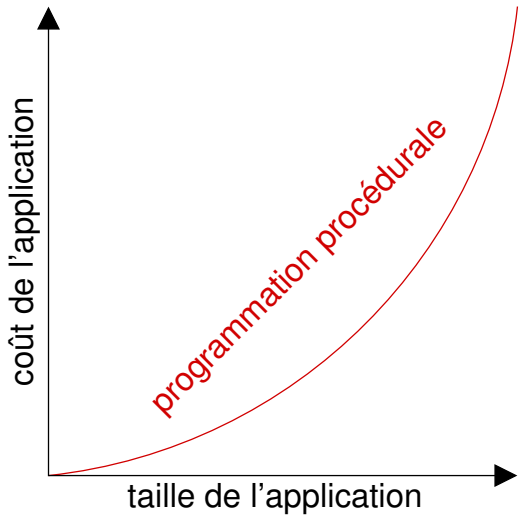
- Le coût des dépenses informatiques $\rightsquigarrow$ principalement coût des logiciels
  - coût du logiciel de plus en plus élevé
  - en programmation procédurale : coût du logiciel croit de manière exponentielle avec la complexité de l'application



Pourquoi la programmation orientée objet ?

- Depuis plusieurs années :
  - le matériel de moins en moins cher, et de plus en plus puissant
  - le développement d'applications de plus en plus performantes et complexes

- Le coût des dépenses informatiques $\rightsquigarrow$ principalement coût des logiciels
  - coût du logiciel de plus en plus élevé
  - en programmation procédurale : coût du logiciel croit de manière exponentielle avec la complexité de l'application



- Objectifs de la programmation orientée objet :
  - diminuer le coût du logiciel
  - augmenter sa durée de vie, sa réutilisabilité et sa facilité de maintenance

Pourquoi la programmation orientée objet ?

- Programmation orientée objet : modélisation directe d'objets du monde réels
  - de très nombreux représentants d'un tout petit nombre de concepts différents
  - exploiter cette redondance $\rightsquigarrow$ petites entités informatiques

Pourquoi la programmation orientée objet ?

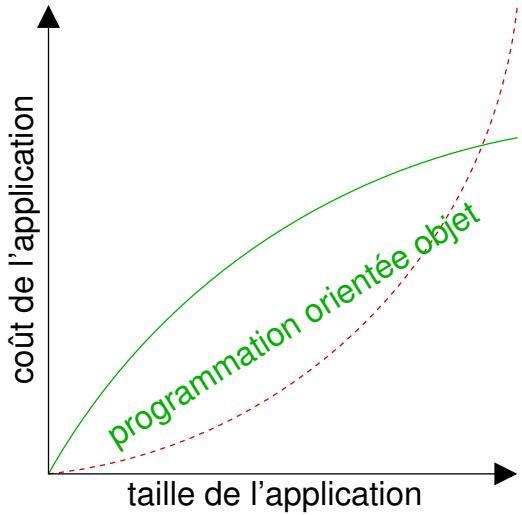
- Programmation orientée objet : modélisation directe d'objets du monde réels
  - de très nombreux représentants d'un tout petit nombre de concepts différents
  - exploiter cette redondance $\rightsquigarrow$ petites entités informatiques

- Réduction de l'impact d'une modification/extension d'une partie d'un programme
  - confinement dans de petites unités qui ont peu de contact avec l'extérieure = objets

Pourquoi la programmation orientée objet ?

- Finalement, un programme objet = production d'un ensemble d'objets
  - séparation de l'interface et de l'implantation de ce que fait l'objet
  - accès à un objet (et à ces données) se fait uniquement via l'interface de haut (fournit aux clients, par exemple)

- Inversion de la courbe d'évolution du coût du logiciel
  - le développement d'une petite application $\rightsquigarrow$ gros effort
  - par contre, son extension $\rightsquigarrow$ effort faible



Plan du cours

La notion d'objet

- Un objet = un concept, une idée ou une entité du monde physique
  - voiture, personne, étudiant, animal, fenêtre graphique, forme géométrique, ...
- Dans un programme, un objet s'apparente à une variable
- Un objet est caractérisé par trois notions
  - les attributs : données de l'objet / variable qu'il contient et représentant son état
  - les méthodes (fonctions membres) : caractérise son comportement, l'ensemble des actions que l'objet peut réalsier, accès/modification des données
  - une identité, qui permet de le distinguer de manière unique des autres objets, indépendamment de son état

La notion de classe

- Une classe = la structure d'un objet
  - elle définit son type
  - déclaration de l'ensemble des entités qui composeront un objet
- Un objet est donc « issu » d'une classe
  - on dit qu'un objet est une instance (ou une occurrence) d'une classe
  - on parle d'*instanciation* de classe

La notion de classe

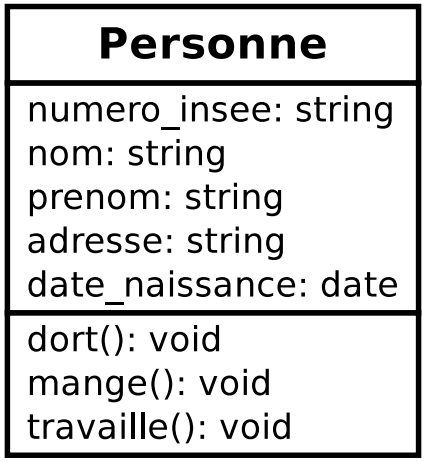
- Une classe = la structure d'un objet
  - elle définit son type
  - déclaration de l'ensemble des entités qui composeront un objet
- Un objet est donc « issu » d'une classe
  - on dit qu'un objet est une instance (ou une occurrence) d'une classe
  - on parle d'*instanciation* de classe
- Une classe est composée de deux parties :
  - les attributs (données membres)
  - les méthodes (fonctions membres)

Exemple de classe et d'objet

- Définition d'une classe Personne

  (on verra plus tard le formalisme utilisé ici)

  - définition d'un type Personne
  - regroupe les propriétés communues aux personnes : caractéristiques (attributs) et comportement (méthodes)

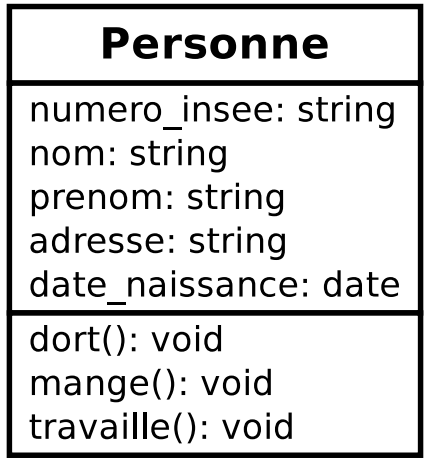


Exemple de classe et d'objet

- Définition d'une classe Personne

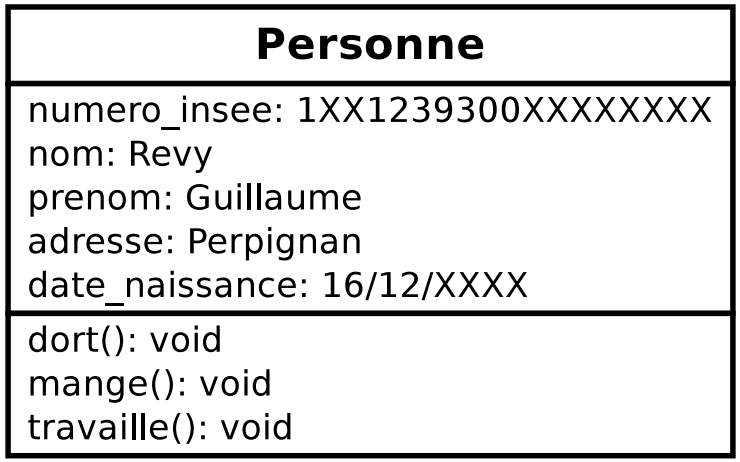
  (on verra plus tard le formalisme utilisé ici)

  - définition d'un type Personne
  - regroupe les propriétés communues aux personnes : caractéristiques (attributs) et comportement (méthodes)



- Instanciation d'un objet de type Personne

  - affectation d'une valeur à chaque attribut
  - possibilité d'agir, via les méthodes



Les trois grands principes de la POO

- Encapsulation $\rightsquigarrow$ rassembler les données et méthodes au sein d'une structure
  - en cachant l'implantation de l'objet (à l'utilisateur, au moins)
  - protection de l'accès aux données $\rightsquigarrow$ mécanisme d'accès / modification
  - augmentation la sécurité d'une application
  - facilité de maintenance de l'application

Les trois grands principes de la POO

- Encapsulation $\rightsquigarrow$ rassembler les données et méthodes au sein d'une structure
  - en cachant l'implantation de l'objet (à l'utilisateur, au moins)
  - protection de l'accès aux données $\rightsquigarrow$ mécanisme d'accès / modification
  - augmentation la sécurité d'une application
  - facilité de maintenance de l'application

- Héritage $\rightsquigarrow$ permettre la création d'une classe à partir d'une classe existante
  - la classe dérivée (*fille*) contient sous certaines conditions les attributs et les méthodes de la classe *mère* (*super-classe*)
  - permet ainsi la réutilisation de code
  - plusieurs types d'héritage : public, protégé, ou privé

Les trois grands principes de la POO

- Encapsulation $\rightsquigarrow$ rassembler les données et méthodes au sein d'une structure
  - en cachant l'implantation de l'objet (à l'utilisateur, au moins)
  - protection de l'accès aux données $\rightsquigarrow$ mécanisme d'accès / modification
  - augmentation la sécurité d'une application
  - facilité de maintenance de l'application

- Héritage $\rightsquigarrow$ permettre la création d'une classe à partir d'une classe existante
  - la classe dérivée (*fille*) contient sous certaines conditions les attributs et les méthodes de la classe *mère* (*super-classe*)
  - permet ainsi la réutilisation de code
  - plusieurs types d'héritage : public, protégé, ou privé

- Polymorphisme $\rightsquigarrow$ permettre de redéfinir dans une classe dérivée les méthodes dont elle hérite de sa classe mère
  - une même méthode possède alors plusieurs formes

Plan du cours

UML - Unified Modeling Language

- UML = langage de modélisation graphique à base de pictogrammes
  - années 90 : fusion des langages de modélisation objet : Booch, OMT, OOSE
  - apparition dans le cadre de la *conception orientée objet*
  - ensemble de diagrammes (de classes, d'objets, ...), permettant de modéliser les aspects statiques et dynamiques d'une application
  - 1997 : normalisation par l'OMG (Object Management Group) $\rightsquigarrow$ 05/2010 : UML 2.3

UML - Unified Modeling Language

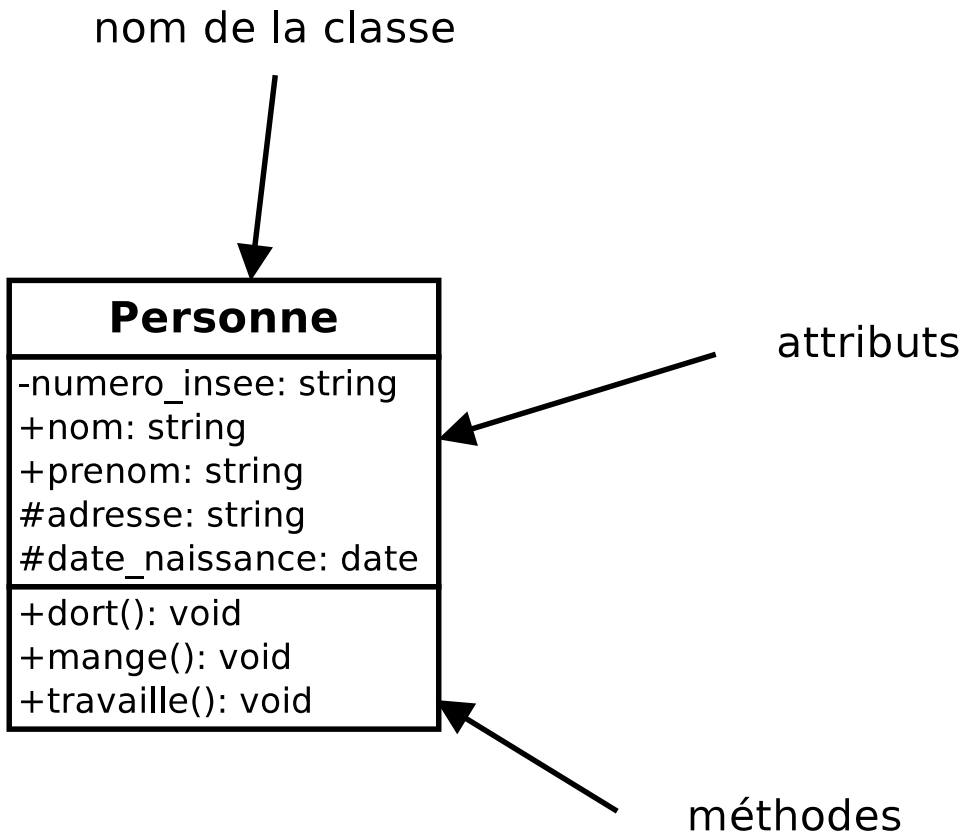
- UML = langage de modélisation graphique à base de pictogrammes
  - années 90 : fusion des langages de modélisation objet : Booch, OMT, OOSE
  - apparition dans le cadre de la *conception orientée objet*
  - ensemble de diagrammes (de classes, d'objets, ...), permettant de modéliser les aspects statiques et dynamiques d'une application
  - 1997 : normalisation par l'OMG (Object Management Group) $\rightsquigarrow$ 05/2010 : UML 2.3

- Dans le cadre de ce cours : utilisation principalement des diagrammes de classes
  - représenter les classes d'un système et les interactions entre elles

Représentation d'une classe

- Reprenons la classe Personne précédente
  - encapsulation $\rightsquigarrow$ définition des méthodes accessibles à l'extérieure de la classe

    = méthodes publiques

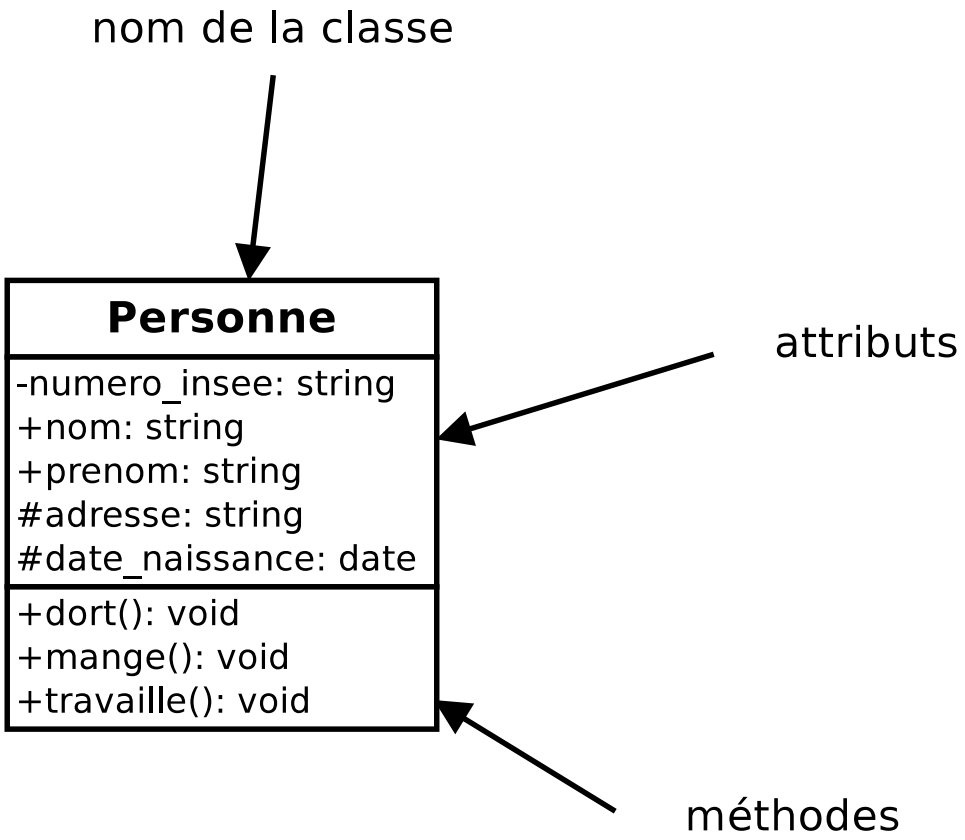


Représentation d'une classe

- Reprenons la classe Personne précédente
  - encapsulation $\rightsquigarrow$ définition des méthodes accessibles à l'extérieure de la classe

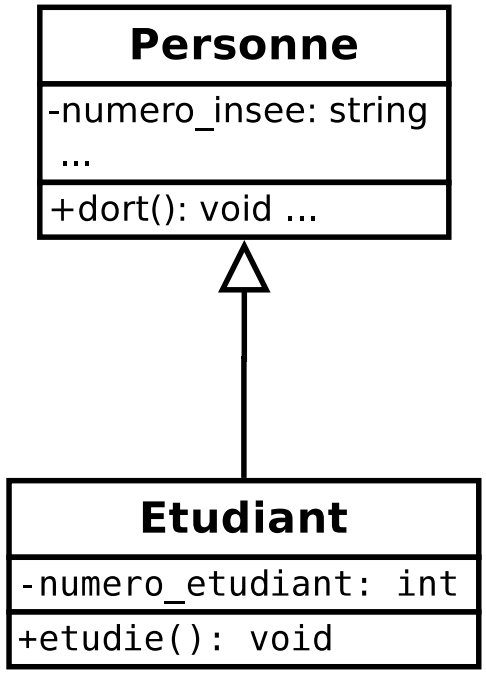
    = méthodes publiques

- Visibilité des attributs / méthodes
  - $-$ privés
  - $\#$ protégés
  - $+$ publics



Relation d'héritage

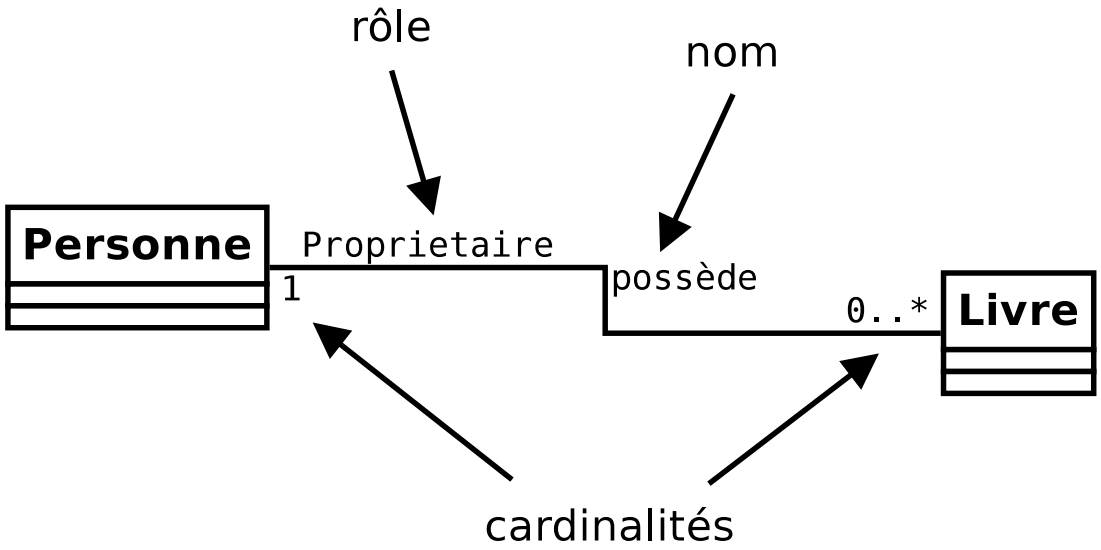
- Intérêt de l'héritage :
  - transfert des propriétés d'une classe mère vers les classes filles
  - généralisation : factorisation de classes en regroupant des propriétés communes
  - spécialisation : ajout d'attributs et méthodes



- Héritage public en C++ $\rightsquigarrow$ le plus utilisé
  - $+$ accessibles par tous
  - $-$ accessibles de la classe elle-même uniquement
  - $\#$ accessibles de la classe elle-même et des classes dérivées

Relation d'association, de composition et d'agrégation

- Association $\rightsquigarrow$ exprime une liaison sémantique bidirectionnelle entre deux classes



- Les cardinalités expriment le nombre d'instances associées
  - un livre appartient à une et une seul personne
  - une personne possède aucun livre ou plusieurs

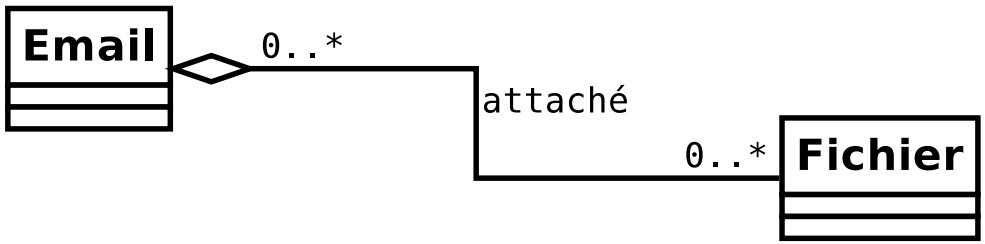
Relation d'association, de composition et d'agrégation

- Agrégation/Composition : relation entre classes, indiquant que les instances d'une classe sont les composants d'une autre

Relation d'association, de composition et d'agrégation

- Agrégation/Composition : relation entre classes, indiquant que les instances d'une classe sont les composants d'une autre

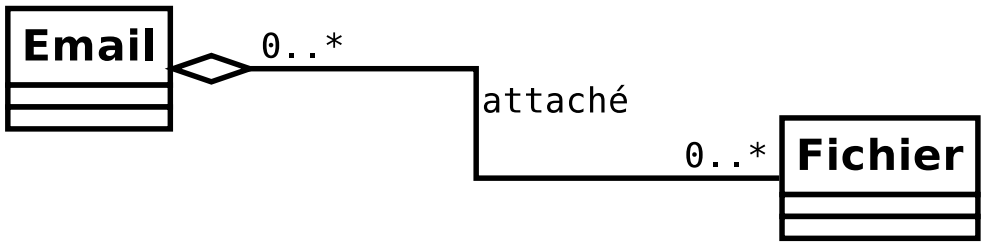
- Agrégation $\rightsquigarrow$ exprime une relation de composition faible
  - les objets agrégés ont une durée de vie indépendante de celle de l'agrégat



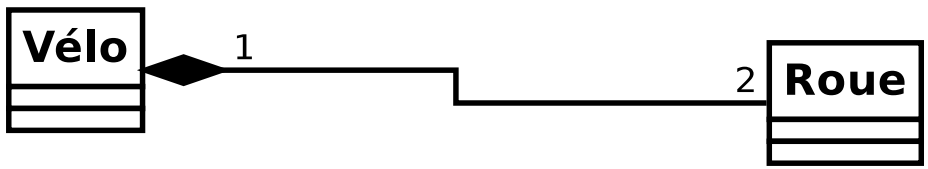
Relation d'association, de composition et d'agrégation

- Agrégation/Composition : relation entre classes, indiquant que les instances d'une classe sont les composants d'une autre

- Agrégation $\rightsquigarrow$ exprime une relation de composition faible
  - les objets agrégés ont une durée de vie indépendante de celle de l'agrégat



- Composition $\rightsquigarrow$ exprime une relation de composition forte
  - les objets agrégés ont une durée de vie dépendante de celle de l'agrégat



Plan du cours

Petite histoire du C++

- Années 80' : mise au point du langage C++ par Bjarne Stroustrup (Bell labs)
- Extension du langage C, mis au point par Ritchie et Kernighan (années 70')
  - ajout de l'approche orientée objet
  - C++ $\rightsquigarrow$ *C with classes* ("C avec des classes")
- 1998 : normalisation par l'ISO (International Organization for Standardization)
  - dernière normalisation : 2003

Fonctionnalités introduites par C++

- Les opérateurs `new` et `delete` pour la gestion d'allocation mémoire
- Les types de données `bool` (booléen), et `string` (chaîne de caractères)
- Le mot clé `const` pour définir des constantes
- Les références
- Les paramètres par défaut dans les fonctions
- Les classes, ainsi que tout ce qui y est lié (héritage, fonctions membres, ...)
- Les référentiels lexicaux (espace de noms) et l'opérateur de résolution `::`
- La surcharge des opérateurs
- Les patrons (ou *templates*)
- La gestion d'exceptions
- ...

Définition d'entités constantes avec le mot clé `const`

- En C++, le mot clé `const` permet à l'utilisateur de définir des entités constantes (fonctions/variables)
  - leur valeur ne pourra pas être modifiée directement
  - elles doivent être initialisées à la déclaration
  - exemple : la plupart des paramètres de fonctions sont lus, mais pas modifiés

```
const int model = 90;          // model est une constante = 90
const int v[] = {1,2,3,4};     // v[i] est une constante

const int x;                   // ERREUR: x doit etre initialise
```

Définition d'entités constantes avec le mot clé `const`

- En C++, le mot clé `const` permet à l'utilisateur de définir des entités constantes (fonctions/variables)
  - leur valeur ne pourra pas être modifiée directement
  - elles doivent être initialisées à la déclaration
  - exemple : la plupart des paramètres de fonctions sont lus, mais pas modifiés

```
const int model = 90;          // model est une constante = 90
const int v[] = {1,2,3,4};     // v[i] est une constante

const int x;                   // ERREUR: x doit etre initialise
```

- Remarque : le mot clé `const` peut modifier le type d'une variable

```
void foo (const int * x) { /* ne peut pas modifier le parametre x */ }

int
main(void){
  int x;
  foo(&x);                  // la variable x peut etre modifiee
}
```

Les pointeurs et les constantes

- Le mot clé `const` peut être utilisé pour :
  - déclarer un pointeur sur un objet constant (`int const * x` ou `const int * x`)
  - déclarer un pointeur constant sur un objet (`int * const x`)

- Exemple 1 : pointeur vers une valeur constante

```
const char * msg = "exemple de message constant";

*msg = 'E'; // ERREUR : modification interdite, car la valeur pointee ne
            //           peut etre modifiee
            // Remarque : *msg ~> msg[0]

msg = "modiciation";    // OK : modification de l'objet pointe, et non de
                        //      sa valeur
```

Les pointeurs et les constantes

- Le mot clé `const` peut être utilisé pour :
  - déclarer un pointeur sur un objet constant (`int const * x` ou `const int * x`)
  - déclarer un pointeur constant sur un objet (`int * const x`)

- Exemple 2 : pointeur constant vers un objet non constant

```
char * const msg = "exemple de pointeur constant";

msg = "Test"; // ERREUR : modification interdit, car l'objet sur lequel
              //          le pointeur pointe ne peut pas etre modifie

*msg = 'E';   // OK : le contenu de l'objet peut etre modifie
              // Remarque : *msg ~> msg[0]
```

Les pointeurs et les constantes

- Le mot clé `const` peut être utilisé pour :
  - déclarer un pointeur sur un objet constant (`int const * x` ou `const int * x`)
  - déclarer un pointeur constant sur un objet (`int * const x`)

- Remarque 1 : L'adresse d'une variable peut être attribuée à un pointeur de constante. Par contre, l'adresse d'une constante ne peut pas être attribuée à un pointeur sur lequel aucune restriction ne s'applique (cette opération autoriserait le changement de la valeur de l'objet).

```
int x = 1;
const int y = 2;

const int * pt1 = &x; // OK
const int * pt2 = &y; // OK

int * pt3 = &y;       // ERREUR : on pourrait sinon modifier y (*pt3 = 17)
```

Les pointeurs et les constantes

- Le mot clé `const` peut être utilisé pour :
  - déclarer un pointeur sur un objet constant (`int const * x` ou `const int * x`)
  - déclarer un pointeur constant sur un objet (`int * const x`)

- Remarque 2 : Un pointeur défini comme pointeur sur une variable `const` ne peut pas être utilisé pour modifier une variable, même si ceux-ci peuvent pointer sur des variables non-`const`.

```
int x = 17;
const int * ptx = &x;

*ptx = 18;              // ERREUR : on ne peut modifier l'objet pointe par ptx,
                        //          bien que x soit modifiable
```

Les références en C++

- En C++, une référénce permet de définir un *alias* sur une variable, fournit un autre nom pour une variable : *X*`&` signifie *référence à X*.
- Une référence s'utilise comme une variable. Pour garantir sa validité, elle doit être initialisée.

```
int x = 17;
int& ref1 = x;          // OK : ref1 et x font maintenant reference au meme int

int& ref2;              // ERREUR : ref2 doit etre initialise
```

Les références en C++

- En C++, une référénce permet de définir un *alias* sur une variable, fournit un autre nom pour une variable : *X*`&` signifie *référence à X*.
- Une référence s'utilise comme une variable. Pour garantir sa validité, elle doit être initialisée.

```
int x = 17;
int& ref1 = x;          // OK : ref1 et x font maintenant reference au meme int

int& ref2;              // ERREUR : ref2 doit etre initialise
```

- Les références sont principalement utilisées pour la spécification des arguments et des valeurs renvoyées pour les fonctions

```
int foo (int& x){ x = x + 1; }

int
main(void){
  int x = 16;
  foo(x);                  // a la sortie de foo : x = 17
}
```

Les références et les constantes

- Une référence non-constante de type *T* & peut être initialisée avec un objet non-constant de type *T* (une variable, par exemple).

```
const int x = 17;
int y = 2011;

int& ref1 = x;          // ERREUR : x n'est pas une variable non-constante
int& ref2 = 17;         // ERREUR : 17 n'est pas un objet non-constant

int& ref3 = y;          // OK : y est une variable non-constante de type int
```

Les références et les constantes

- Une référence non-constante de type *T* `&` peut être initialisée avec un objet non-constant de type *T* (une variable, par exemple).

```
const int x = 17;
int y = 2011;

int& ref1 = x;          // ERREUR : x n'est pas une variable non-constante
int& ref2 = 17;         // ERREUR : 17 n'est pas un objet non-constant

int& ref3 = y;          // OK : y est une variable non-constante de type int
```

- Une référence constante (ou `const`-référence) de type *const T* `&` peut être initialisée avec un objet non-constant de type *T* ou constant de type *const T*.

```
const int x = 17;
int y = 2011;

const int& ref1 = x;    // OK : x est pas une variable constante (const int)
const int& ref2 = 17;   // OK : 17 est pas un objet constant

const int& ref3 = y;    // OK : y est une variable non-constante de type int
```

Plan du cours

Modélisation d'un point du plan 2D

- On souhaite modéliser un point du plan 2D
  - un point est caractérisé par ses coordonnées (x,y) dans le plan
  - et il peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.

Modélisation d'un point du plan 2D

- On souhaite modéliser un point du plan 2D
  - un point est caractérisé par ses coordonnées (x,y) dans le plan
  - et il peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.
- On doit pouvoir également :
  - l'initialiser,
  - et accéder / modifier ses coordonnées.

Modélisation d'un point du plan 2D

- On souhaite modéliser un point du plan 2D
  - un point est caractérisé par ses coordonnées (x,y) dans le plan
  - et il peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.

- On doit pouvoir également :
  - l'initialiser,
  - et accéder / modifier ses coordonnées.

- Remarque
  - attributs = privés
  - méthodes = publiques

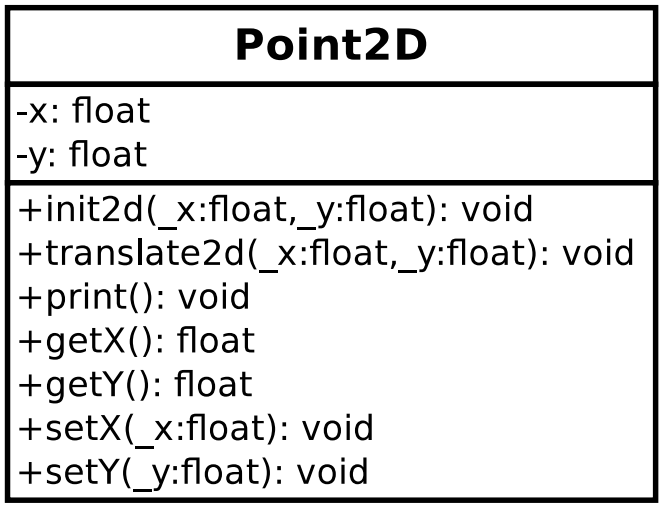
Point2D
-x: float -y: float
+init2d(_x:float,_y:float): void +translate2d(_x:float,_y:float): void +print(): void +getX(): float +getY(): float +setX(_x:float): void +setY(_y:float): void

Comment déclarer une classe ?

```
// Classe.hpp

#ifndef __CLASSE_HPP__  // <-- directives de precompilation
#define __CLASSE_HPP__  //

class Classe : <private|protected|public> ClasseMere
{

private:

  // declaration des attributs et methodes prives

protected:

  // declaration des attributs et methodes proteges

public:

  // declaration des attributs et methodes publics

};

#endif // __CLASSE_HPP__
```

- Grâce aux directives de précompilation, la classe n'est incluse qu'une seule fois

Déclaration de la classe Point2D



- Pour respecter le principe d'encapsulation $\rightsquigarrow$ attributs privés (voire protégés)
  - 1 des 3 grands principes de la POO
  - prévoir des méthodes `get` et `set`

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;

public:

  void init2d(float, float);
  void translate2d(float, float);

  void print(void);

  float getX(void) const;
  float getY(void) const;

  void setX(float);
  void setY(float);

};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

Comment définir les méthodes d'une classe ?

- On peut définir une méthode directement dans la définition de la classe

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{
  // ...

  float getX(void) const{ return x; }
  // ...
};

#endif // __POINT2D_HPP__
```

- On parle de fonction inline
  - ⊕ son utilisation améliore le temps d'exécution des programmes (en réduisant la lourdeur de l'appel aux fonctions)
  - ⊖ son utilisation peut entraîner une augmentation de la taille du code de l'application
    - à n'utiliser que pour des fonctions de petite taille

Comment définir les méthodes d'une classe ?

- Ou bien de manière séparée :

```
// Classe.cpp

typeDeRetour Classe::nomDeLaMethode( <liste des parametres> ){

  // definition de la methode

}
```

Comment définir les méthodes d'une classe ?

- Ou bien de manière séparée :

```
// Classe.cpp

typeDeRetour Classe::nomDeLaMethode( <liste des parametres> ){

  // definition de la methode

}
```

- Reprenons par exemple la méthode `translate2d` de la classe Point2D :

```
// Point2D.cpp

void Point2D::translate2d(float _x, float _y){
  x += _x;
  y += _y;
}
```

Définition de la classe Point2D

```
// Point2D.cpp

#include <iostream>          // iostream -> gestion des entrees-sorties
#include "Point2D.hpp"

void Point2D::init2d(float _x, float _y){
  x = _x;
  y = _y;
}

void Point2D::translate2d(float _x, float _y){
  x += _x;
  y += _y;
}

void Point2D::print(void){
  std::cout << " --> Point2D (" << x << "," << y << ")" << std::endl;
}

float Point2D::getX(void) const{ // idem : float Point2D::getY(void) const{ ... }
  return x;
}

void Point2D::setX(float _x){     // idem : void Point2D::setY(float _y){ ... }
  x = _x;
}
```

Utilisation simple de la classe Point2D

- Déclaration de deux points
  - un point statique $\rightsquigarrow$ (2.5,1.5)
  - un point dynamique $\rightsquigarrow$ (17.3,15.1)

```
// c1-expl1.cpp

#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D a;            Point2D * b = new Point2D(); // allocation dynamique de b

  a.init2d(2.5,1.5);    b->init2d(17.3,15.1);
  std::cout << "Avant translate2d(1,1)" << std::endl;
  a.print();            b->print();

  a.translate2d(1,1);   b->translate2d(1,1);
  std::cout << "Apres translate2d(1,1)" << std::endl;
  a.print();            b->print();
                        delete b;                 // liberation de la memoire de b
  return 0;
}
```

Compilation et exécution de cet exemple simple

- Utilisation du compilateur GNU C++ (g++ $\geq$ 4.5)
  - compilation séparée
  - idéalement (notamment dans vos TP/projet), utilisation d'un makefile

```
$> g++ -c Point2D.cpp
$> g++ -c c1-expl1.cpp
$> g++ -o c1-expl1 c1-expl1.o Point2D.o

$> ./c1-expl1
Avant translate(1,1)
--> Point2D (2.5,1.5)
--> Point2D (17.3,15.1)
Apres translate(1,1)
--> Point2D (3.5,2.5)
--> Point2D (18.3,16.1)
```

Intérêt des différents niveaux de visibilité

- Rappel : dans la classe Point2D,
  - les attributs x et y sont privés,
  - et les méthodes getX() et getY() sont publiques.

```
// c1-expl2.cpp

#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D a; a.init2d(2.5,1.5);

  std::cout << "Acces a l'attribut x" << std::endl;
  std::cout << "  valeur de a.x: " << a.x << std::endl;          // ERREUR
  //
  // Point2D.hpp: In function 'int main()':
  // Point2D.hpp:11:9: error: 'float Point2D::x' is private
  //    float x, y;
  //          ^
  // c1-expl2.cpp:13:41: error: within this context
  //    std::cout << "  valeur de a.x: " << a.x << std::endl;  // ERREUR
  //
  std::cout << "  valeur de a.x: " << a.getX() << std::endl;    // OK

  return 0;
}
```

Schéma de compilation séparée

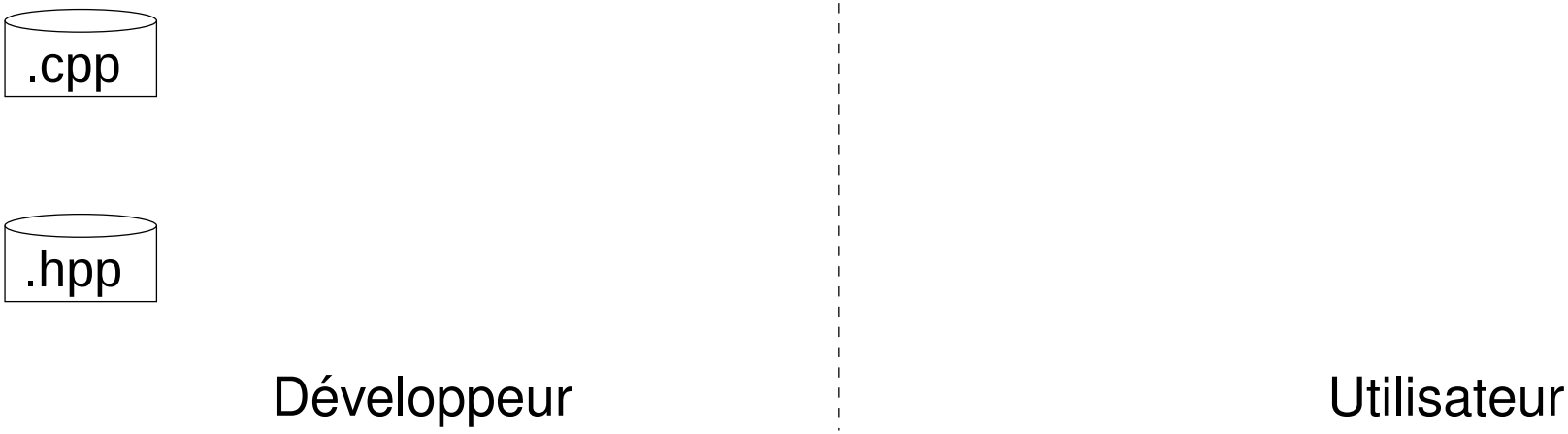
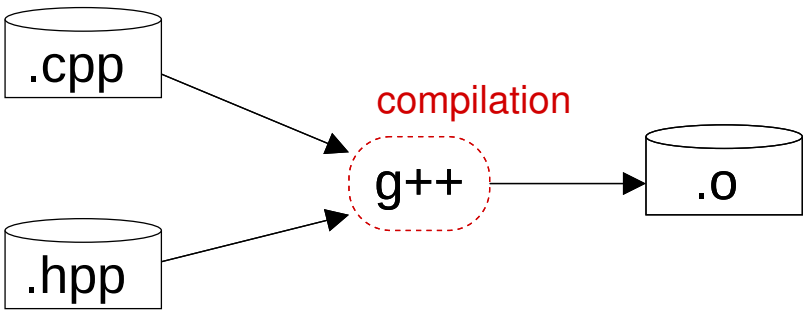


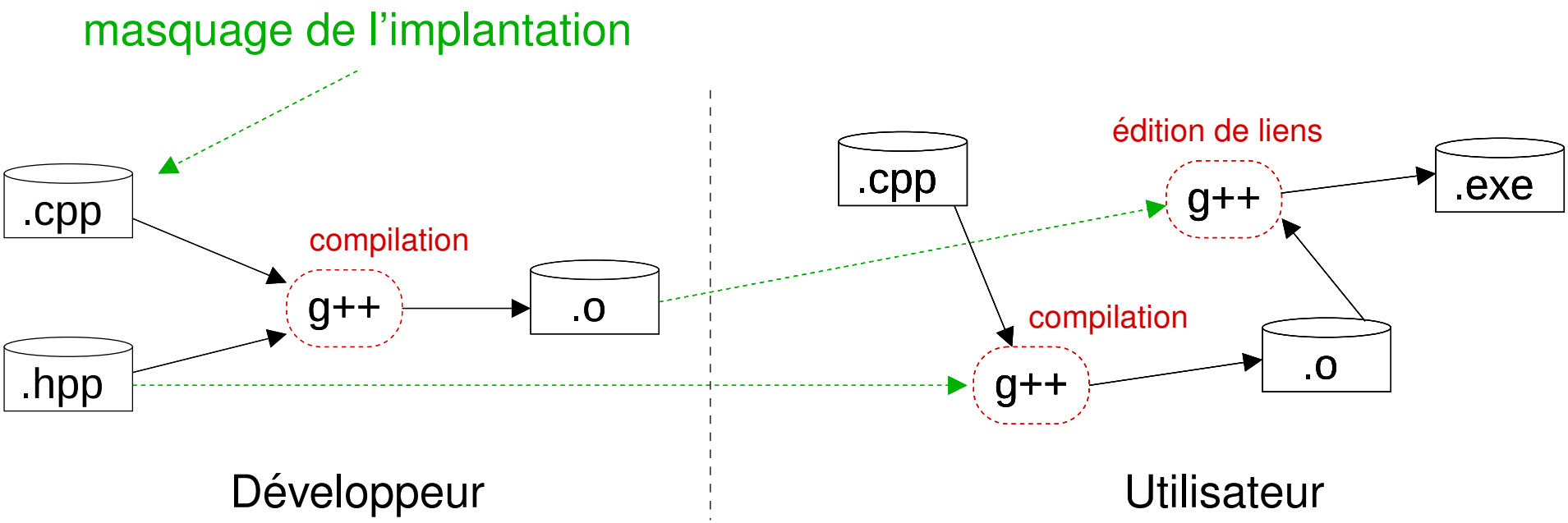
Schéma de compilation séparée



Développeur

Utilisateur

Schéma de compilation séparée



Questions ?