Programmation Orientée Objet

Composition de classes, classes et fonctions amies, et héritages

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Plan du cours

Plan du cours

Les constructeurs et destructeurs

- Un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - appelé immédiatement à chaque fois qu'un espace est alloué pour un objet
  - constructeur par défaut, par copie, ou spécialisé (défini par l'utilisateur)
- Un destructeur $\rightsquigarrow$ méthode spéciale automatiquement appelée lors de la destruction d'une instance de classe
  - juste avant que la mémoire utilisée par l'objet ne soit récupérée par le système
  - il y a au plus un destructeur par classe

Les constructeurs et destructeurs

- Un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - appelé immédiatement à chaque fois qu'un espace est alloué pour un objet
  - constructeur par défaut, par copie, ou spécialisé (défini par l'utilisateur)
- Un destructeur $\rightsquigarrow$ méthode spéciale automatiquement appelée lors de la destruction d'une instance de classe
  - juste avant que la mémoire utilisée par l'objet ne soit récupérée par le système
  - il y a au plus un destructeur par classe

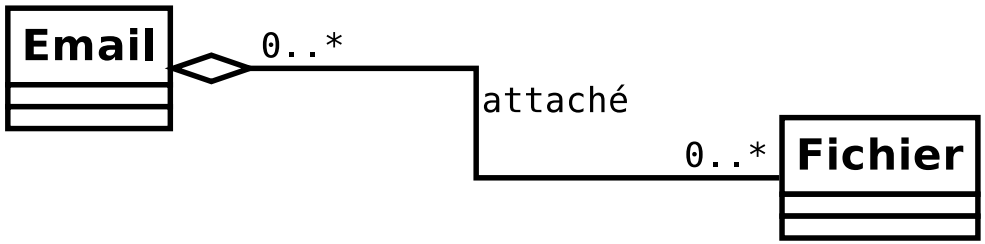
```
// ClasseA.cpp
class ClasseA
{
public:
  ClasseA();                  // constructeur par defaut
  ClasseA(ClasseA const&);    // constructeur par copie
  ClasseA(...);               // constructeur specialise
  ~ClasseA();                 // destructeur
};
```

Plan du cours

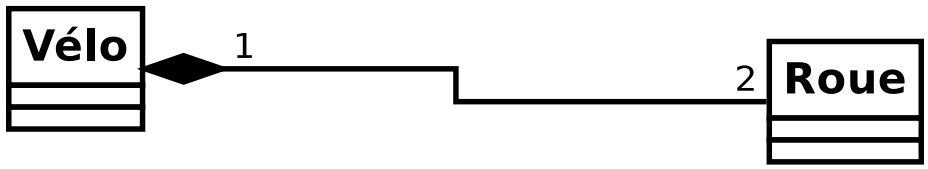
Rappel sur les relations de composition et d'agrégation

- 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



Agrégation : exemple d'une classe Email

- On souhaite modéliser un email, à travers la classe Email, pouvant être composé d'un fichier joint (classe Fichier)

```
#ifndef __EMAIL_HPP__
#define __EMAIL_HPP__

#include "Fichier.hpp"

class Email
{
private:
  Fichier * f;

public:
  Email();
  Email(Fichier *);
  ~Email();
};

#endif
```

```
#include <iostream>
#include "Email.hpp"

Email::Email()
{ std::cout << "Creation d'un email sans piece
     jointe" << std::endl; }

Email::Email(Fichier * f)
{
  this->f = f;
  std::cout << "Creation d'un email avec piece
       jointe" << std::endl;
}

Email::~Email()
{ std::cout << "Suppression d'un email... mais
     pas de la piece jointe" << std::endl; }
```

Agrégation : exemple d'une classe Email

```
// c3-expl0.cpp
#include <iostream>
#include "Email.hpp"
#include "Fichier.hpp"

int
main( void )
{
  Fichier * f = new Fichier();     // Appel du constructeur de la classe Fichier
  Email e(f);                      // Appel du constructeur de la classe Email
  std::cout << "J'envoie mon email..." << std::endl;
  delete f;                        // Appel du destructeur de la classe Fichier
  return 0;                        // Appel du destructeur de la classe Email
}
```

Agrégation : exemple d'une classe Email

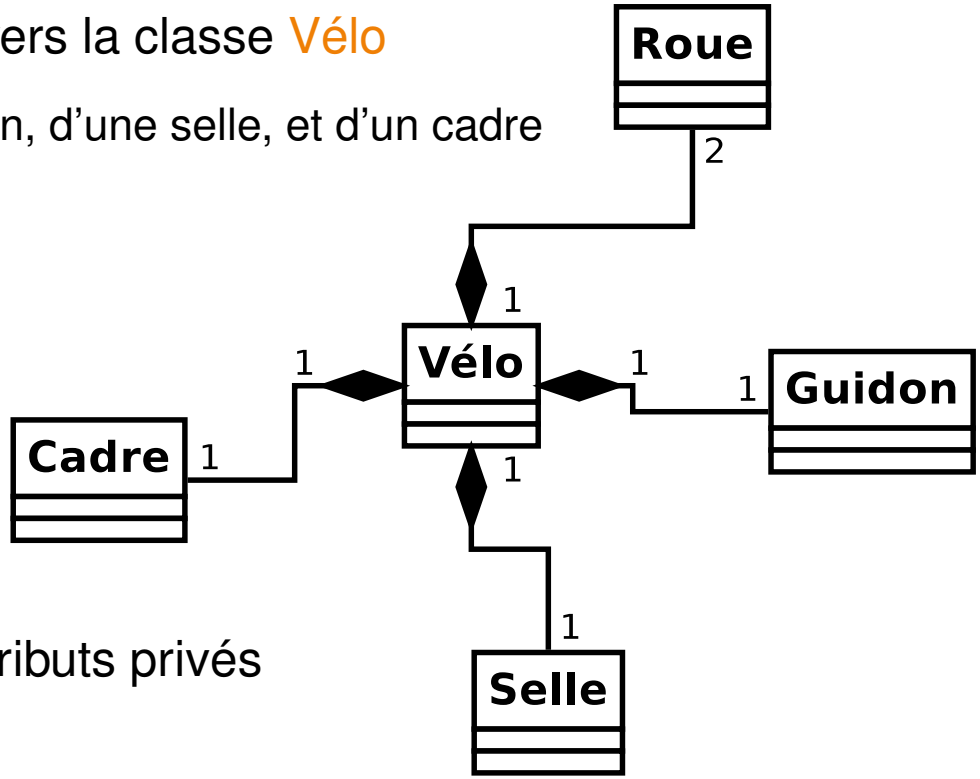
```
// c3-expl0.cpp
#include <iostream>
#include "Email.hpp"
#include "Fichier.hpp"

int
main( void )
{
  Fichier * f = new Fichier();     // Appel du constructeur de la classe Fichier
  Email e(f);                      // Appel du constructeur de la classe Email
  std::cout << "J'envoie mon email..." << std::endl;
  delete f;                        // Appel du destructeur de la classe Fichier
  return 0;                        // Appel du destructeur de la classe Email
}
```

```
$> ./c3-expl1
Creation d'un fichier a joindre a un
     mail
Creation d'un email avec piece jointe
J'envoie mon email...
Suppression d'un fichier
Suppression d'un email... mais pas de
     la piece jointe
```

- Construction et destruction des composants effectués par l'utilisateur
- Respect de la spécification de l'agrégation

Composition : exemple d'une classe Vélo

- On souhaite modéliser un vélo, à travers la classe Vélo
  - composé de deux roues, d'un guidon, d'une selle, et d'un cadre



- On définit une classe Velo, avec 5 attributs privés
  - 2 objets de type Roue
  - 1 objet de type Guidon
  - 1 objet de type Selle
  - 1 objet de type Cadre

Composition : exemple d'une classe Vélo

```
#ifndef __VELO_HPP__
#define __VELO_HPP__

#include "Roue.hpp"
#include "Guidon.hpp"
#include "Selle.hpp"
#include "Cadre.hpp"

// Declaration de la classe Velo
class Velo
{
private:
  Roue rAvant, rArriere;
  Guidon g;
  Selle s;
  Cadre c;

public:
  Velo();
  ~Velo();
};

#endif
```

```
#include <iostream>
#include "Velo.hpp"

// Constructeur de la classe Velo
Velo::Velo()
{
  std::cout << "Construction d'un velo" <<
    std::endl;
}

// Destructeur de la classe Velo
Velo::~Velo()
{
  std::cout << "Destruction d'un velo" <<
    std::endl;
}
```

Composition : exemple d'une classe Vélo

```
// c3-expl1.cpp
#include <iostream>
#include "Velo.hpp"

int
main( void )
{
  Velo v;
  std::cout << "J'utilise maintenant mon velo..." << std::endl;
  return 0;
}
```

Composition : exemple d'une classe Vélo

```
// c3-expl1.cpp
#include <iostream>
#include "Velo.hpp"

int
main( void )
{
  Velo v;
  std::cout << "J'utilise maintenant mon velo..." << std::endl;
  return 0;
}
```

```
$> ./c3-expl1
Construction d'une roue
Construction d'une roue
Construction d'un guidon
Construction d'une selle
Construction d'un cadre
Construction d'un velo
J'utilise maintenant mon velo...
Destruction d'un velo
Destruction d'un cadre
Destruction d'une selle
Destruction d'un guidon
Destruction d'une roue
Destruction d'une roue
```

- Construction des composants puis de l'objet composé, et destruction dans le sens inverse
  - appels automatiques par le constructeur/destructeur du composé
- Respect de la spécification de la composition

Plan du cours

Qu'est ce qu'une fonction amie ?

- Une fonction *amie* d'une classe A est une fonction qui, sans être membre de la classe A, a le droit d'accès à tous les membres de la classe, c'est-à-dire, les membres publics, protégés et privés.
  - doit être déclarée dans la classe qui accorde les droits $\rightsquigarrow$ ici, la classe A
  - indifféremment dans la partie publique, protégée ou privée
  - sa déclaration doit être précédée du mot clé `friend`

```
// A.hpp
class A
{
  private:
    int a;
  public:
    A();
    friend void print(A&);
};
```

```
// A.cpp
A::A(){ a = 0; }

void print(A const & obj_a)
{
  std::cout << obj_a.a << std::endl;
}
```

- Remarque : bien que définie à l'extérieure de la classe A, la méthode `print` a accès aux attributs (notamment, aux attributs privés) de la classe A

Remarques sur les fonctions amies

- Remarques :
  - elle n'est attachée à aucun objet particulier
  - le pointeur `this` n'y est donc pas défini

- Exemple : opération arithmétique entre deux matrices

```
// Matrix.hpp
class Matrix
{
  private:
    // ...
  public:
    Matrix();
    friend Matrix& multiplication(Matrix&, Matrix&);
};
```

```
// Matrix.cpp
void multiplication(Matrix& a, Matrix& b){ /* ... */ }
```

Qu'est ce qu'une classe amie ?

- Une classe *amie* d'une classe A est une classe qui a le droit d'accès à tous les membres de la classe A, c'est-à-dire, les membres publics, protégés et privés.
  - doit être déclarée dans la classe qui accorde les droits $\rightsquigarrow$ ici, la classe A
  - indifféremment dans la partie publique, protégée ou privée
  - sa déclaration doit être précédée du mot clé `friend`

```
// A.hpp
class A
{
  private:
    int a;
  public:
    A();
    friend class B;
};
```

```
// B.hpp
class B
{
  private:
    A * obj;
  public:
    B(){ obj = new A(); }
    void acces_to_a(){
      std::cout << obj->a << std::endl;
    }
};
```

- Remarque : la classe B a accès aux attributs privés de la classe A $\rightsquigarrow$ seuls les objets B ont le droit d'accéder aux attributs de A, le reste du système manipule des objets B.

Remarques générales sur la relation d'amitié

- La relation d'*amitié* doit être utilisée avec précaution $\rightsquigarrow$ c'est une entorse à l'ensemble des règles qui régissent les droits d'accès.
- La relation d'*amitié* n'est pas transitive $\rightsquigarrow$ les *amis* des *amis* de la classe A ne sont pas forcément les *amis* de la classe A.
- Le caractère *ami* n'est pas hérité.
- Les *amis* de classes imbriquées n'ont pas de droit particulier sur les membres de la classe de niveau supérieur.

Plan du cours

Principe de l'héritage

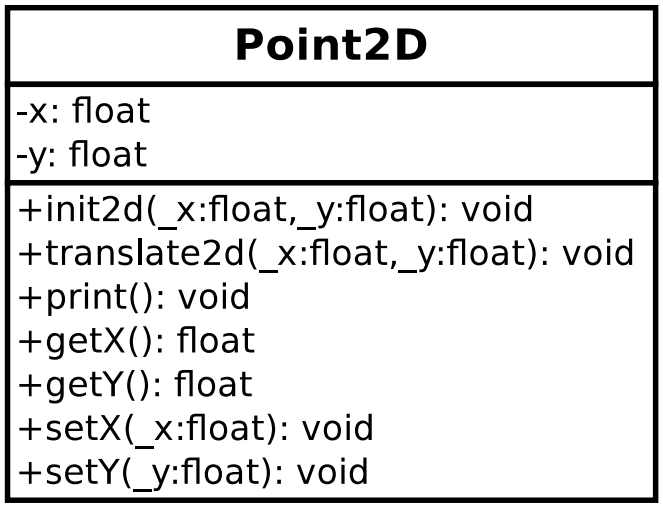
- Mécanisme d'héritage $\rightsquigarrow$ définition d'une classe (appelée *classe dérivée* / *fille*) par assemblage des membres (attributs et méthodes) d'une ou plusieurs classes (*classes de bases directes*), et des membres (attributs et méthodes) spécifiques de la nouvelle *classe dérivée*.
  - 1 des 3 grands principes de la POO

```
// Classe.hpp

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

class Classe : <private|protected|public> ClasseMere1,
               <private|protected|public> ClasseMere2, ...
{
private:
  // ...
protected:
  // ...
public:
  // ...
};

#endif // __CLASSE_HPP__
```

Principe de l'héritage

- Mécanisme d'héritage $\rightsquigarrow$ définition d'une classe (appelée *classe dérivée* / *fille*) par assemblage des membres (attributs et méthodes) d'une ou plusieurs classes (*classes de bases directes*), et des membres (attributs et méthodes) spécifiques de la nouvelle *classe dérivée*.
  - 1 des 3 grands principes de la POO
- Intérêt de l'héritage :
  - réutilisation d'une classe telle quelle dans une autre, par composition
  - extension ou adaptation d'une classe à partir d'une autre
- L'héritage peut être
  - simple $\rightsquigarrow$ une seule classe de base directe
  - multiple $\rightsquigarrow$ plusieurs classes de base directes

Retour, encore une fois, à la déclaration de la classe Point2D



- Pour respecter le principe d'encapsulation $\rightsquigarrow$ attributs privés, voire protégés
  - 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 étendre à un point de l'espace 3D ?

- Un point de l'espace 3D
  - est caractérisé par ses coordonnées (x,y,z) dans l'espace
  - et peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.

Comment étendre à un point de l'espace 3D ?

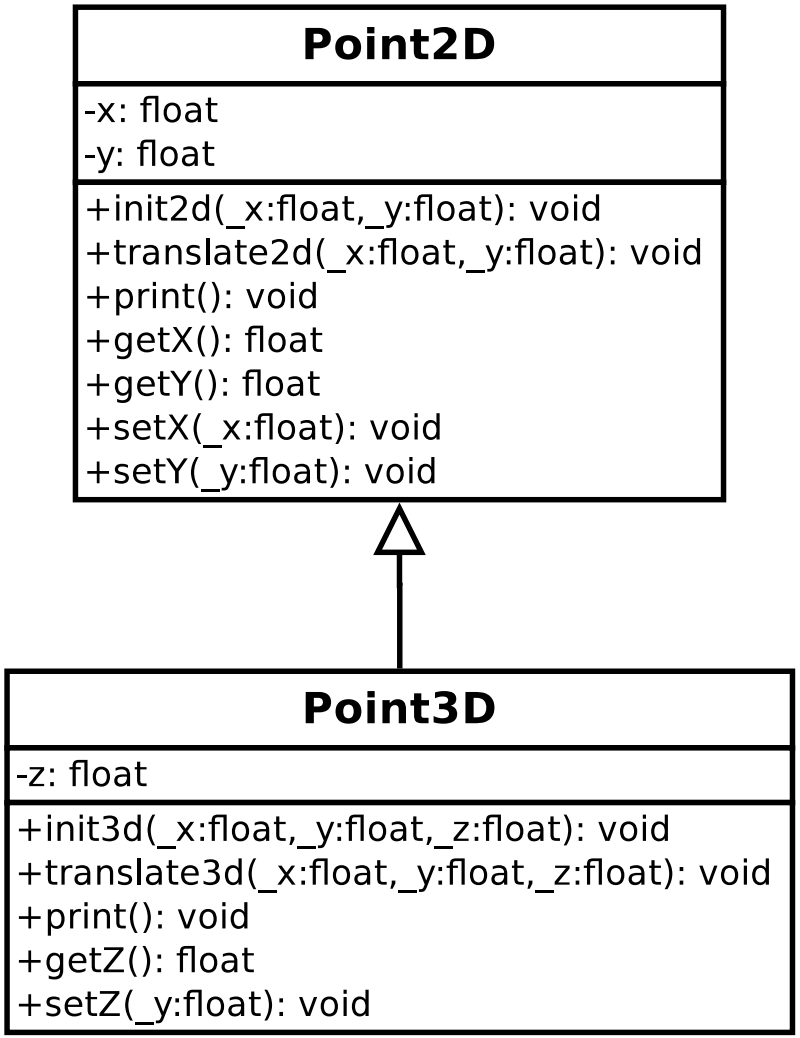
- Un point de l'espace 3D
  - est caractérisé par ses coordonnées (x,y,z) dans l'espace
  - et peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.
- On peut voir un point 3D comme une extension d'un point 2D
  - ajout d'une coordonnée en z, et de ses méthodes d'accès/modification
  - redéfinition des méthodes d'initialisation et de translation
  - et spécialisation de la méthode d'affichage

Comment étendre à un point de l'espace 3D ?

- Un point de l'espace 3D
  - est caractérisé par ses coordonnées (x,y,z) dans l'espace
  - et peut être affiché (au moins ses coordonnées, dans un premier temps) et translaté.
- On peut voir un point 3D comme une extension d'un point 2D
  - ajout d'une coordonnée en z, et de ses méthodes d'accès/modification
  - redéfinition des méthodes d'initialisation et de translation
  - et spécialisation de la méthode d'affichage
- Finalement, on peut utiliser l'héritage public
  - extension de la classe Point2D
  - définition d'une classe Point3D, qui hérite de Point2D

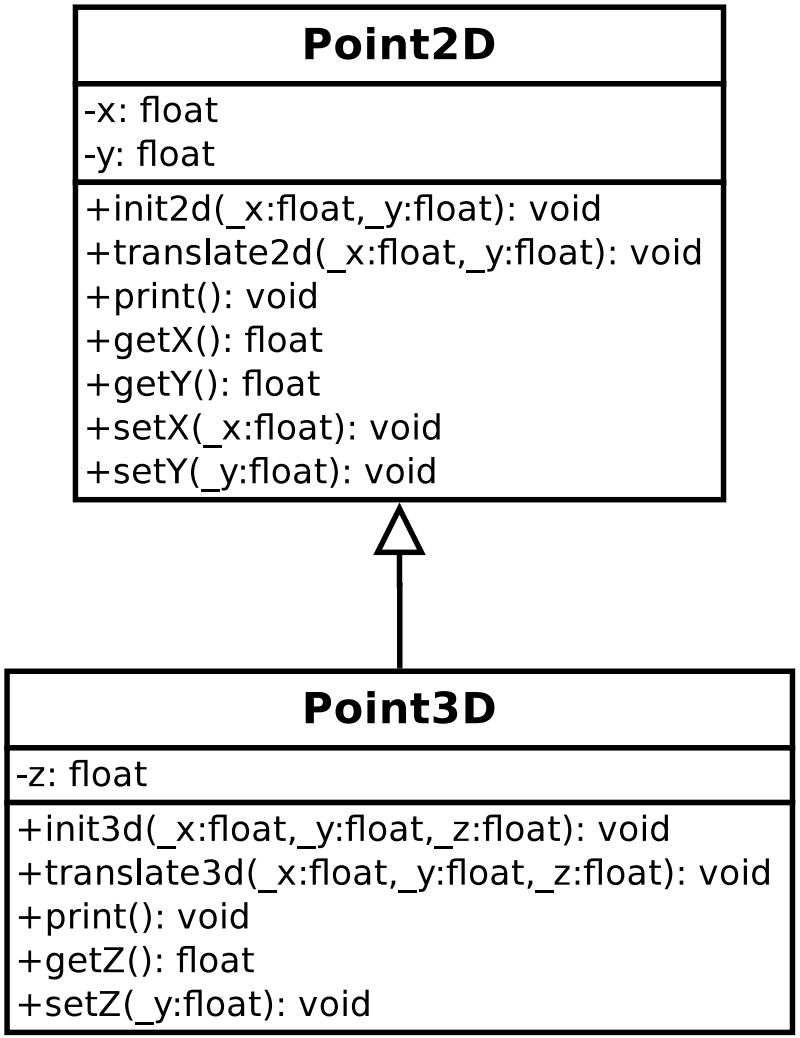
Comment étendre à un point de l'espace 3D ?

- Dans la classe Point3D
  - ajout d'une coordonnée en *z*
  - et des méthodes associées
  - définition des méthodes `init3d` et `translate3d`
  - et spécialisation de la méthode d'affichage `print`



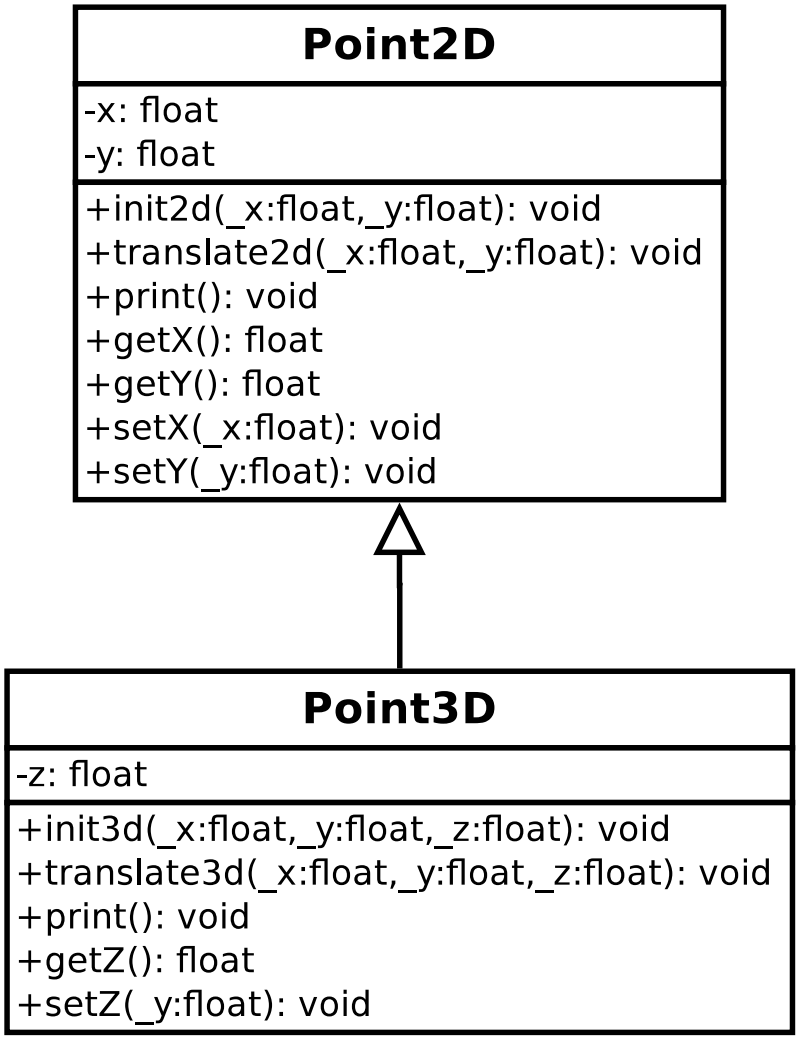
Comment étendre à un point de l'espace 3D ?

- Dans la classe Point3D
  - ajout d'une coordonnée en *z*
  - et des méthodes associées
  - définition des méthodes `init3d` et `translate3d`
  - et spécialisation de la méthode d'affichage `print`
- Les attributs et les méthodes sont hérités de la classe Point2D

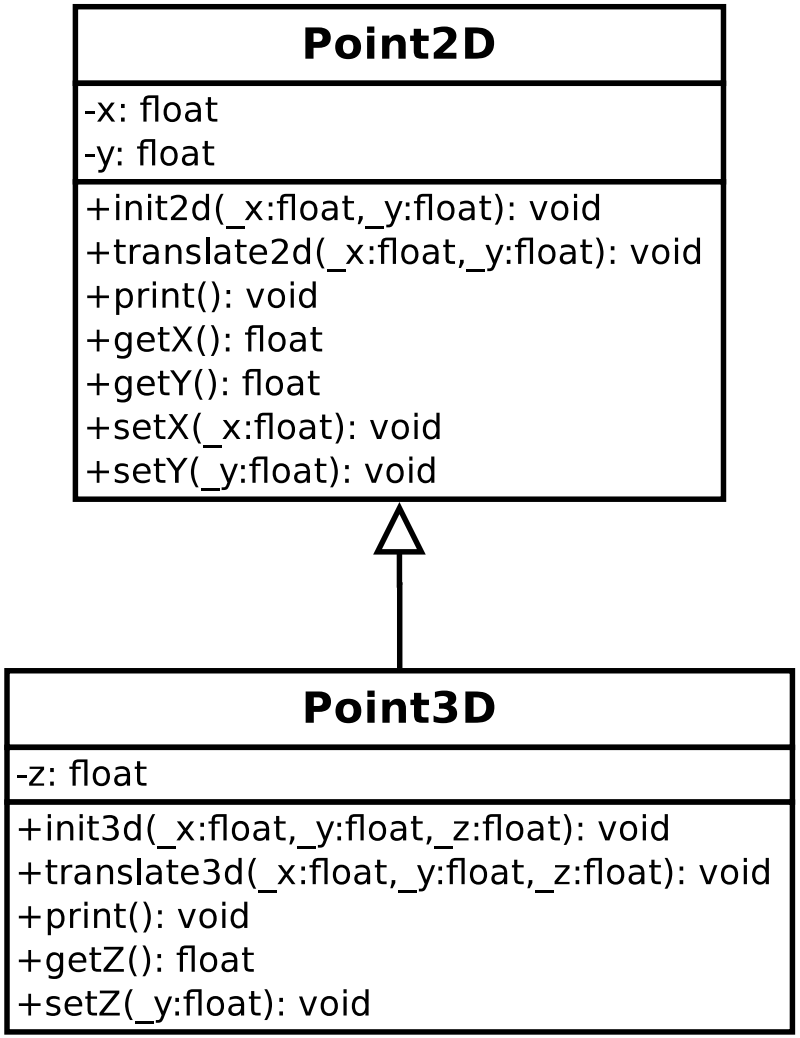


Comment étendre à un point de l'espace 3D ?

- Dans la classe Point3D
  - ajout d'une coordonnée en *z*
  - et des méthodes associées
  - définition des méthodes `init3d` et `translate3d`
  - et spécialisation de la méthode d'affichage `print`
- Les attributs et les méthodes sont hérités de la classe Point2D
- Mais les attributs `x` et `y` ne sont pas accessibles depuis Point3D
  - car déclarés en "zone privée"



Extension à la classe Point3D




```cpp
// Point3D.hpp

#ifndef __POINT3D_HPP__
#define __POINT3D_HPP__

#include "Point2D.hpp"

class Point3D : public Point2D
{

private :

  float z;

public:

  void init3d(float, float, float);
  void translate3d(float, float, float);

  void print(void);

  float getZ(void) const;

  void setZ(float);

};

#endif // __POINT3D_HPP__
```

Définition de la classe Point3D

```
// Point3D.cpp

#include <iostream>          // iostream -> gestion des entrees-sorties
#include "Point3D.hpp"

void Point3D::init3d(float _x, float _y, float _z){
  setX(_x);                          // on ne peut pas modifier x et y directement,
  setY(_y);                          // car on n'y a pas acces
  z = _z;
}

void Point3D::print(void){
  std::cout << " --> Point3D (" << getX() << "," << getY() << "," << z << ")";
  std::cout << std::endl;            // on n'a pas acces directement a x et y
}

void Point3D::translate3d(float _x, float _y, float _z){
  translate2d(_x,_y);                // on appelle la methode de la classe mere
  z += _z;
}

float Point3D::getZ(void) const{ return z; }

void Point3D::setZ(float _z){ z = _z; }
```

Finalement, utilisation de la classe Point3D

```
// c1-expl2.cpp

#include <iostream>
#include "Point2D.hpp"
#include "Point3D.hpp"

int
main( void )
{
  Point3D a;     a.init3d(1,1,1);     // a = (1,1,1)

  std::cout << "Avant translate(0.17,0.34,0.51)" << std::endl; a.print();

  a.translate3d(0.17,0.34,0.51);
  std::cout << "Apres translate(0.17,0.34,0.51)" << std::endl; a.print();

  std::cout << "X : " << a.getX() << std::endl; // <-- acces aux methodes
  std::cout << "Y : " << a.getY() << std::endl; // <-- de Point2D
  std::cout << "Z : " << a.getZ() << std::endl;

  return 0;
}
```

Héritage et accessibilité des membres

- Suivant le mode d'héritage (privé, protégé ou public), les membres d'une classe *mère* sont accessibles sous certaines conditions dans les classes dérivées.

- Accessibilité des zones de la classe *mère* dans la classe *fille* :

		Zone ... de la classe de base		
		privée	protégée	publique
Héritage	public	inaccessible	protégé	public
	protégé	inaccessible	protégé	protégé
	privé	inaccessible	privé	privé

- Remarques
  - héritage le plus utilisé $\rightsquigarrow$ public
  - héritage par défaut $\rightsquigarrow$ privé

Exemple (1) d'accessibilités

```
// ...
class A
{
public:
  int x;
  int getX(){ return x; }
};

class PublicA: public A
{
public:
  int printX(){ std::cout << x << std::endl; }
};
```

```
int
main( void )
{
  A obja;
  PublicA objpa;

  objpa.printX();                          //  printX() a acces a l'attribut x de A
  std::cout << obja.x << std::endl;        //  l'attribut x est public dans A
                                           //     -> la fonction main() y a acces
  std::cout << objpa.x << std::endl;       //  l'attribut x est public dans PrivateA
                                           //     -> la fonction main() y a acces
}
```

Exemple (2) d'accessibilités

```
// ...
class A
{
protected:
  int x;
public:
  int getX(){ return x; }
};

class PublicA: public A
{
public:
  int printX(){ std::cout << x << std::endl; }
};
```

```
int
main( void )
{
  A obja;
  PublicA objpa;

  objpa.printX();                          //  printX() a acces a l'attribut x de A
  std::cout << obja.x << std::endl;        //  ERREUR : l'attribut x est protege dans A
                                           //     -> la fonction main() n'y a pas acces
  std::cout << objpa.x << std::endl;       //  ERREUR : l'attribut x est protege dans
                                           //     PublicA
                                           //     -> la fonction main() n'y a pas acces
}
```

Redéfinition des fonctions membres

- Lorsqu'un membre de la classe dérivée a le même nom qu'un membre de la classe de base, le premier masque le second, quelque soit le mode d'héritage
  ⇝ exemple : méthode avec la même signature, ...

```
// ...
class A
{
public:
  void printX(){ std::cout << "A" << std::endl; }
};

class PublicA: public A
{
public:
  int printX(){ std::cout << "PublicA" << std::endl; }
};
```

```
int
main( void )
{
  A obja;
  PublicA objpa;

  obja.printX();   //  printX() de la classe A -> "A"
  objpa.printX();  //  printX() de la classe PublicA -> "PublicA"
}
```

Constructeur et héritage

- Lors de la création d'un objet, une série de constructeurs est appelée
  - initialisation de chaque classe de la chaîne d'héritage
- Ordre des appels : classe mère (*super-classe*) $\rightsquigarrow$ classe dérivées

```
#ifndef __A_HPP__
#define __A_HPP__

class A
{
public:
  A();
};

class B: public A
{
public:
  B();
};

class C: public B
{
public:
  C();
};

#endif // __A_HPP__
```

```
#include "A.hpp"

A::A(){ std::cout << "Constructeur de A"
     << std::endl; }
B::B(){ std::cout << "Constructeur de B"
     << std::endl; }
C::C(){ std::cout << "Constructeur de C"
     << std::endl; }
```

```
#include <iostream>
#include "A.hpp"

int
main( void )
{
  C objc; // ...
  return 0;
}
```

Constructeur et héritage

- Lors de la création d'un objet, une série de constructeurs est appelée
  - initialisation de chaque classe de la chaîne d'héritage
- Ordre des appels : classe mère (*super-classe*) $\rightsquigarrow$ classe dérivées

```
#ifndef __A_HPP__
#define __A_HPP__

class A
{
public:
  A();
};

class B: public A
{
public:
  B();
};

class C: public B
{
public:
  C();
};

#endif // __A_HPP__
```

```
#include "A.hpp"

A::A(){ std::cout << "Constructeur de A"
     << std::endl; }
B::B(){ std::cout << "Constructeur de B"
     << std::endl; }
C::C(){ std::cout << "Constructeur de C"
     << std::endl; }
```

```
$> ./c3-expl2
Constructeur de A
Constructeur de B
Constructeur de C
```

Initialisation des classes de base

- L'initialistion des instances des classes de bases se fait sur le même modèle que l'initialisation des membres constants $\rightsquigarrow$ avant l'appel du constructeur
  - certains attributs peuvent être privés

```
// Point3D.hpp

#ifndef __POINT3D_HPP__
#define __POINT3D_HPP__

#include "Point2D.hpp"

class Point3D : public Point2D
{

private :

  float z;

public:

  // Constructeurs
  Point3D();
  Point3D(float, float, float);
  Point3D(Point3D const&); // ...

};

#endif // __POINT3D_HPP__
```

```
// Point3D.cpp

#include <iostream>
#include "Point3D.hpp"

Point3D::Point3D():Point2D(0,0){ z = 0; }

Point3D::Point3D(float _x, float _y,
                 float _z):Point2D(_x,_y)
                     { z = _z; }

Point3D::Point3D(Point3D const& pt):
  Point2D(pt.getX(),pt.getY())
{
  // >> il faut declarer Point2D::getX()
  //    et Point2D::getY() en 'const'
  z = pt.z;
}
```

Destructeur et héritage

- Lors de la destruction d'un objet, une série de destructeurs est appelée
  - nettoyage de chaque classe de la chaîne d'héritage
- Ordre des appels : classe dérivées $\rightsquigarrow$ classe mère (*super-classe*)

```
#ifndef __A_HPP__
#define __A_HPP__

class A
{
public:
  ~A();
};

class B: public A
{
public:
  ~B();
};

class C: public B
{
public:
  ~C();
};

#endif // __A_HPP__
```

```
#include "A.hpp"

A::~A(){ std::cout << "Destructeur de A"
     << std::endl; }
B::~B(){ std::cout << "Destructeur de B"
     << std::endl; }
C::~C(){ std::cout << "Destructeur de C"
     << std::endl; }
```

```
#include <iostream>
#include "A.hpp"

int
main( void )
{
  C objc; // ...
  return 0;
}
```

Destructeur et héritage

- Lors de la destruction d'un objet, une série de destructeurs est appelée
  - nettoyage de chaque classe de la chaîne d'héritage
- Ordre des appels : classe dérivées $\rightsquigarrow$ classe mère (*super-classe*)

```
#ifndef __A_HPP__
#define __A_HPP__

class A
{
public:
  ~A();
};

class B: public A
{
public:
  ~B();
};

class C: public B
{
public:
  ~C();
};

#endif // __A_HPP__
```

```
#include "A.hpp"

A::~A(){ std::cout << "Destructeur de A"
     << std::endl; }
B::~B(){ std::cout << "Destructeur de B"
     << std::endl; }
C::~C(){ std::cout << "Destructeur de C"
     << std::endl; }
```

```
$> ./c3-expl2
Destructeur de C
Destructeur de B
Destructeur de A
```

Questions ?