

# POO avec C++ 2011

Xavier JUVIGNY

ONERA

January 23, 2018

# Plan du cours

1 Introduction à l'orienté objet

2 POO en C++

# Orienté Objets

## Programmation orienté objets

- Un style de programmation et une façon de programmer
  - Avec un langage traditionnel;
  - Avec un langage orienté objets

Programmer avec un langage POO ne fait pas de l'orienté objets

## Définition de l'objet

- Un objet contient à la fois des données et des fonctions faisant des traitements sur ces données.
- Autres concepts similaires aux autres méthodes de programmation.
- Un jargon à connaître :

Type	Variable	Valeurs
Classe	Variable	Valeurs/Objets
<code>class</code> Vecteur;	<code>int</code> u; <code>double</code> x;	[1.,3.,5.], "Coucou", 23, Vecteur(1.,3.,5.,7.)

# Service vs réalisation

## Objectif

- Une valeur qui représente un nuage de points : Un ensemble de couples  $X$  et  $Y$ , réels
- De nombreuses réalisations sont possibles. Par exemple

## Exemple d'interface

- Un vecteur de  $X$ , un autre de  $Y$ , même taille
- Un vecteur de couples  $(X, Y)$
- Avec un dictionnaire : Chaque point avec une clef  $(i, j)$  et le couple  $(x, y)$  en valeur ou un vecteur de  $X$  et un vecteur de  $Y$  avec une clef par zone, par exemple les  $X > 0$ , les  $X < 0$  et les  $X == 0$
- Pour chaque mise en œuvre...
  - Une fonction qui retourne les  $X$ , une les  $Y$
  - Donne le nombre de points
  - Une qui donne tous les points, au dessus/au dessous d'un  $X$  ou d'un  $Y$ , avec une valeur exacte de  $X$

# La donnée pilote l'algorithme

## Exemple avec choix de réalisation n°1

```
auto coords=std::make_pair(std::vector<double>{100},std::vector<double>{100});
using coordinates=decltype(coords);
auto& y = coords.second;
auto yLowerThan( const coordinates& crds, double val ) {
    coordinates c; std::vector<int> r;
    for ( int i = 0; i < crds.first.size(); ++i ) {
        if (crds.second[i] < val ) {
            c.first.push_back(crds.first[i]); c.second.push_back(crds.second[i]);
            r.push_back(i);
        }
    }
    return std::make_tuple(c,r); }
```

## Exemple avec choix de réalisation n°2

```
using coordinates=std::vector<std::pair<double,double>>;
coordinates coords(100);
std::vector<double> y(100); int iy=0;
for ( const auto& c : coords ) y[++iy] = c.second;
auto yLowerThan( const coordinates& crds, double val ) {
    coordinates c; std::vector<int> r;
    for ( int i = 0; i < crds.size(); ++i ) {
        if (crds[i].second < val ) {
            c.push_back(crds[i]); r.push_back(i);
        }
    }
    return std::make_tuple(c,r); }
```

# Concrétisation de l'interface

## Homogénéité de l'interface

- Divers objets possèdent la même interface :
  - Des objets de même type : réels, entiers, etc.
  - Les listes, les tableaux, les dictionnaires, etc :

```
std::vector<double> u{ {3.4, 5.8, 1.2} };  
std::list<double> l{ {3.4, 5.8, 1.2} };  
std::array<3,double> a{ {3.4, 5.8, 1.2} };
```

```
for ( auto& val : u) val *= 0.5;  
for ( auto& val : l) val *= 0.5;  
for ( auto& val : a) val *= 0.5;
```

```
u.push_back(2.);  
l.push_back(2.);
```

# Mise en œuvre d'une interface

## Mise au point de l'interface

Méthode de développement TDD ( Test Driven Development ) :

- L'interface se conçoit en écrivant les tests de "haut niveau" en premier lieu;
- On regarde avec le client si l'interface proposé dans les tests correspond à son attente.

## Mise au point d'une identité vecteur géométrique

```
#include "vecteur.hpp"

Geometry::Vecteur u{ 1., 0., 0. } };
Geometry::Vecteur v;
v.x = u.z; v.y = -u.x; v.z = u.y

std::cout << u << " + " << v << " = " << u+v << std::endl;
double nrmSq_u = u.normalize(); double nrmSq_v = v.normalize();
Geometry::Vecteur w = u - (u|v)*v;
```

En premier lieu, bien définir son interface

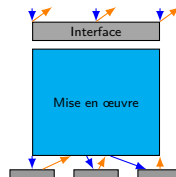
## Conception d'une classe

- Concevoir une classe n'est pas un fin en soi
- C'est un moyen de réaliser une interface ( mais on peut utiliser d'autres techniques si elles sont plus adaptées ).

# Conception d'une interface

## Interface-1

- Que la partie utilisable de l'objet :
  - Accès aux données pertinentes de l'objet;
  - Des constantes;
  - Des fonctions ( méthodes ) pour manipuler l'objet;
  - Gestion du comportement;
- Un exemple : Le téléphone





# Conception d'une interface (2)

## Fonctions

- `status:int initialize()`
- `value:float currentData(variable:string)`
- `terminate()`

Plusieurs variables pour `currentData` ?

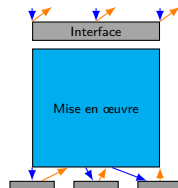
## Constantes

- `variables = ['Temperature', 'Pressure']`

## Comportement

- créer l'objet
- `initialize` (commence l'échantillonnage)
- `currentData` (si `status` est `True`)
- `terminate` (arrêt échantillonnage)
- détruire l'objet

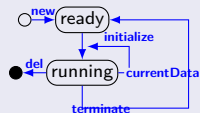
Quid gestion des erreurs ? Bad initialization, Timeout, ...



# Conception d'une interface (3)

## Comportement

Généralement décrit par un automate.



## Interface N+1

Interface utilisant les services offerts par l'interface.

**Exemple :** Interface visualisation données envoyées par capteurs.

## Interface N-1

Interface utilisée ( non unique ) par l'interface actuelle

- Se traduit en C++ par un `#include`
- Ou par un appel système

# Principes de l'OO

## L'interface

Les services offerts à l'extérieur du module/bibliothèque

## L'encapsulation

Isoler/Cacher les choix de mises en œuvres

## La factorisation

Factoriser les services communs au sein de types "abstraits"

- Polymorphisme
- *Héritage*

## Mises en œuvre

Réalisable dans tous les langages de programmation.  
Les langages POO fournissent des facilités.

# Encapsulation

## Une unité

Un paquet comprenant tout ce qu'on peut trouver pour installer une bibliothèque/module qui contient :

- des données;
- des traitements sur les données;
- mais aussi :
  - des constantes;
  - traitement des erreurs;
  - des tests
  - de la documentation
  - la production

## Mise en œuvre

Le choix de ce qui est dans l'objet ou pas est arbitraire. Il dépend de l'application. Il n'est pas possible de définir un objet idéal pour toutes les applications. C'est donc différent d'une modélisation mathématique.

# OO – Principes – Isolation

## Une interface

Les services, les fonctions, les constantes,...

## Une réalisation

- La structure interne des valeurs manipulées;
- La mise en œuvre effective des services.

## La règle d'or

### Ne pas exposer la réalisation

```
double retrieveCurrentMax( const Capteur& a ) {  
    return a.max;  
}
```

```
double retrieveCurrentMax( const Capteur& a ) {  
    return std::min(a.max, CONST_VAL_MAX);  
}
```

```
double retrieveCurrentMax( const Capteur& a ) {  
    a.update(); return a.max;  
}
```

```
double retrieveCurrentMax( const Capteur& a ) {  
    return CONST_VAL_MAX;  
}
```

# OO – Principes – Factorisation

## Si deux objets sont similaires

- Interfaces avec intersection non nulle;
- Possible de factoriser cette interface.

## Les matrices

- Une matrice inversible est une matrice carrée;
- Une matrice carrée est une matrice;
- Une matrice est un tableau à deux dimensions.

## Règle à suivre

Si **B** est un sous-ensemble de **A**, alors **b** est au moins un **a**.

*Mais... Si je peux rajouter une ligne à une matrice, que doit faire une matrice carrée ?*

# Principes – Factorisation – 2

## Deux grandes règles de base pour les classes

- La factorisation se conçoit sur les objets, puis se formalise sur les classes;
- La factorisation est arbitraire : elle dépend de l'application visée.

## Erreurs traditionnelles de la conception des classes

- Il ne s'agit pas de modéliser le monde;
- Il s'agit de réaliser du code pour une application.

# OO - Principes – Factorisation - - 4

## Règle de base de l'OO

**L'encapsulation** et son corollaire immédiat et essentiel : **l'interface**

## La factorisation

*Matrice inversible, matrice carrée, matrice,...*

- La factorisation des interfaces donne la hiérarchie des classes;
- Factoriser seulement les valeurs utilisables;
- Généralisation
- À ne pas confondre avec la **composition**.

## La dérivation

*Matrice inversible, matrice symétrique inversible*

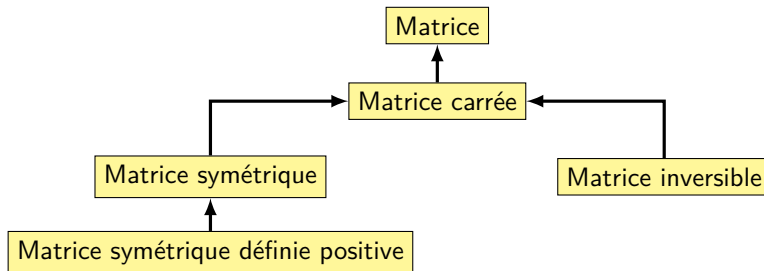
- Ne dériver que les valeurs utilisables;
- Spécialisation



# Hiérarchie des classes

## Factorisation/Dérivation des types

- Forme une arborescence
  - Une matrice carrée est une matrice;
  - Une matrice symétrique est une matrice carrée;
  - Une matrice symétrique définie positive est une matrice symétrique;
  - Une matrice inversible est une matrice carrée;



# Syntaxe de déclaration de la classe

## Déclaration

```
class Vecteur //<--- Déclaration d'un nouveau type
{
public: // Partie accessible par une fonction externe à la classe
    Vecteur() = default; // Constructeur par défaut
    void setAsPoint(); // Méthode publique de la classe
    double x,y,z; // Attributs publics de la classe
private: // Partie privée accessible uniquement aux méthodes de la classe.
    bool is_point() const; // Méthode privée de la classe
    double w; // Attributs privés de la classe
};

Vecteur u; // <-- Nouvelle variable de type Vecteur
           //      faisant appel au constructeur par défaut
u.x = 3.0; // Accès aux attributs publics de la classe
u.y = 2.0;
u.z = 1.0;
u.setAsPoint();
```

On a défini ici un constructeur par défaut qui initialise par défaut chaque attribut : pour les réels, l'initialisation ne fait que réserver la mémoire pour les trois attributs.

# Les constructeurs

## Les constructeurs

- Un constructeur décrit la façon d'initialiser une variable d'un type donné;
- Il doit être nommé du même nom que le type;
- Il peut y avoir plusieurs constructeurs possédant une liste d'arguments différents;

```
class Vecteur {  
public:  
    Vecteur(); // Constructeur par défaut  
    // Initialisation du vecteur à partir d'une liste de valeurs  
    Vecteur( const std::initializer_list<double>& init_vals );  
    Vecteur( double cx, double cy, double cz ); // Initialisation à partir de 3 entiers  
    Vecteur( double cx ); // Initialisation à partir de 1 réel  
    Vecteur( const Vecteur& u ); // Constructeur de copie  
    Vecteur( Vecteur&& u ); // Constructeur de déplacement  
    ... // D'autres constructeurs si on le souhaite...  
};  
  
Vecteur u; // Appel au constructeur par défaut  
Vecteur v{ 1., 3., -5. }; // Appel construction avec liste d'initialisation  
Vecteur w( 3., 4., 2. ); // Appel constructeur avec trois réels  
Vecteur z(1.); // Appel constructeur avec un réel  
Vecteur y(z); // Appel le constructeur de copie  
Vecteur w(std::move(y)); // Appel constructeur de déplacement
```

# Les constructeurs

## Règles du C++ sur les constructeurs

- Si aucun constructeur n'est déclaré, C++ définit un constructeur par défaut sans paramètres, appelant les constructeurs par défaut de chacun des attributs de la classe;
- Si non déclaré, C++ définit un constructeur de copie et un constructeur de déplacement qui appellent les constructeurs de copie/déplacement de chacun des attributs;
- Si un des attributs est non copiable ( respectivement non déplaçable ), alors le constructeur de copie ( respectivement de déplacement ) est automatiquement supprimé;
- Il est possible d'appeler un constructeur dans la liste d'initialisation ( voir plus loin ) d'un autre constructeur de la même classe.

Pour éviter de mauvaises surprises, toujours déclarer les trois constructeurs : celui par défaut ( sans paramètre ), celui de copie et celui de déplacement, quitte à les déclarer par défaut ou supprimés ( voir transparent suivant ).

# Les constructeurs par défaut/supprimés

## Constructeurs par défaut

Il suffit de déclarer le constructeur sans paramètres, de copie ou de déplacement en spécifiant à la fin qu'on prend celui défini par défaut par C++ :

## Suppression d'un constructeur déclaré par défaut

Si on veut interdire le constructeur de copie, ou sans paramètre, ou de déplacement, il suffit de déclarer ce constructeur avec un suffixe `delete` pour le supprimer

```
class Vecteur {  
public:  
    Vecteur() = default; // Constructeur sans paramètre par défaut  
    Vecteur( const Vecteur& u ) = default; // Construction de copie par défaut  
    Vecteur( Vecteur&& u ) = delete; // Interdiction de déplacer  
    ...  
private: // Partie de la classe uniquement accessible aux membres de la classe.  
    std::array<double,3> m_valeurs;  
};
```

# Mise en œuvre d'un constructeur

## Constructeur en ligne

On peut définir un constructeur dans la déclaration de la classe. Il est alors en ligne (inline) si pas trop gros.

```
// vecteur.hpp
class Vecteur {
    ...
    //              initialisation des attributs      ----
    //                                              V
    Vecteur( const std::initializer_list<double>& vals ) : m_valeurs{vals} {}
    Vecteur( double x, double y, double z );//<--- seulement déclarée !
    ...
private:
    std::array<double,3> m_valeurs;
};
```

La définition du constructeur doit y être brève !

## Mise en œuvre d'un constructeur

Définition du constructeur dans le fichier de mise en œuvre ( .cpp ) :

```
// vecteur.cpp
# include "vecteur.hpp"
// Pour toutes méthodes/attributss définies en dehors de la classe :
// Nom de la classe :: Nom de la méthode/attribut
Vecteur::Vecteur( double x, double y, double z ) : m_valeurs{} {
    m_valeurs[0] = x; m_valeurs[1] = y; m_valeurs[2] = z;
}
```

# Constructeur de copie/de déplacement

## Constructeur de copie

- Pour instancier un nouvel objet en copiant les données d'un objet existant;
- Utilisé implicitement lors du passage par valeur d'un objet en paramètre, lors du retour par valeur d'une fonction d'une variable non locale à la fonction

```
class Vecteur {  
    ...  
    Vecteur( const Vecteur& u ) : m_arr_coefs(u.m_arr_coefs) {}  
    ...  
};
```

## Constructeur de déplacement

- Instancie un nouvel objet en déplaçant les données d'un objet existant vers ce nouvel objet. Après création du nouvel objet, l'ancien objet ne possède plus ces données initiales;
- Utilisé lors du retour par valeur d'une fonction d'une variable locale à cette fonction.

```
class Vecteur {  
    ...  
    Vecteur( Vecteur&& u ) : m_arr_coefs(std::move(u.m_arr_coefs))  
    {}  
};
```

# Destructeur

## Le destructeur

- Définit la façon dont un objet de la classe va se détruire.
- Peut être défini par défaut : appel de destructeur correspondant à chacun des attributs de l'objet;
- Est nommé du même nom que la classe précédé d'un ~.
- Un seul destructeur par classe. Ne possède pas de paramètres;
- Appelé au pire lorsqu'on sort du bloc de visibilité de l'objet.

```
class Vecteur { ...  
    ~Vecteur() = default;  
};
```



# Gestion des exceptions pour les constructeurs (+)

## Gestion des exceptions lors de la construction

Utiliser le bloc `try... catch` autour de l'initialisation de l'objet :

```
class Matrix {  
public:  
    ...  
    Matrix( int nrows, int ncols );  
    ...  
private:  
    std::vector<double> m_arr_coefs;  
};  
  
Matrix::Matrix( int nrows, int ncols ) try :  
    m_arr_coefs(nrows*ncols) // <--- Liste d'initialisation  
    { // Corps du constructeur.  
        ...  
    }  
catch(std::bad_alloc& err)  
{  
    ...  
    throw std::bad_alloc(errMsg);  
}
```

- Impossible de rattraper une exception au sein d'un constructeur;
- Rattraper une exception pour un constructeur ne peut servir qu'à le reinterpréter et le renvoyer.
- L'objet n'existe pas lors du traitement de l'exception

# Gestion des erreurs dans un constructeur(+)

## Exceptions

- Seul moyen propre de signaler les erreurs dans un constructeur;
- Si une exception est levée, l'objet ne sera **jamais créé**;
- L'objet est créé uniquement au moment de quitter le constructeur;
- Exception dans sous-objet ( attribut/héritage ) : objet non construit;
- Pour une sous-partie optionnelle, utiliser l'idiome de mise en œuvre **Pimpl**.

```
// CapteurPression.hpp
class CapteurPression {
    ...
private:
    PressureSensor m_pressure;
    ThermalSensor m_thermal ;
    class OptionalSensors;
    OptionalSensors* m_pt_other_sensors;
};
// CapteurPression.cpp
class CapteurPression::OptionalSensors {
public:
    Clock m_clock;
    HydroSensor m_hydrosensor;
};
...
```

# Exercice : nuage de points

## Construction d'un nuage de point

- 1 Construire une classe `CloudOfPoints` permettant d'instancier des nuages de  $N$  points en 2D,  $N$  pouvant varier d'un nuage à l'autre;
- 2 Mettre des affichages au début de chaque constructeur et destructeur pour tracer l'appel des constructeurs;
- 3 Tester votre classe à l'aide du programme suivant :

```
# include "CloudOfPoints.hpp"
int main()
{
    CloudOfPoints nuage1{ 10 };
    CloudOfPoints nuage2{ nuage1 };
    CloudOfPoints nuage3;
    CloudOfPoints nuage4{2, 5, nuage1}; // Copie les points 2 à 5 du nuage1.
}
```

- 4 Bonus : Gérer les erreurs lors de la construction d'un nuage à l'aide des exceptions;
- 5 Bonus : Pouvoir construire un nuage de la manière suivante :

```
CloudOfPoints nuage5{ Point{1.,0.,0.},Point{1.,1.,0.},Point{1.,1.,1.} };
```

où `Point` est un alias sur `std::array<double,3>`

# Les méthodes

## Les méthodes

- Fonctions membres de la classe appelables via un objet;
- Peuvent accéder "directement" aux parties publiques et privées de l'objet appelant la méthode;
- Ou au travers d'un pointeur spécial `this` pointant sur l'objet ayant appelé la méthode;

```
class Vecteur {  
    ...  
    double normalize(); // <--- Déclaration d'une méthode  
    ...  
};  
...  
double Vecteur::normalize() { // <--- Définition de la méthode  
    double nrm = std::sqrt(m_valeurs[0]*m_valeurs[0] + m_valeurs[1]*m_valeurs[1] +  
                           m_valeurs[2]*m_valeurs[2] );  
    m_valeur[0] /= nrm; m_valeurs[1] /= nrm; m_valeurs[2] /= nrm;  
    return nrm;  
}  
...  
Vecteur u;  
...  
u.normalize(); // <--- u appelle la méthode normalize
```

# Méthode avec qualificateur const

## Qualification const d'une méthode

- Rajout du mot clef `const` à la fin de la méthode à la déclaration et à la définition;
- Garantit que l'objet ne sera pas modifié ( vu de l'extérieur ) en appelant cette méthode;
- Rajout de `mutable` à un attribut pour le modifier dans une méthode `const`;
- Les attributs mutables doivent ne pas modifier l'objet vu de l'extérieur ( à travers l'interface ).

```
class Vecteur { ...  
    double normL2() const; // Calcul de la norme L2  
    ...  
private: ...  
    mutable double m_proxy_norm; // <-- Conserve la norme calculée.  
                                // Si coefficient modifié, on remet cette norme à -1  
};  
  
double Vecteur::sqNormL2() const {  
    if (m_proxy_norm < 0.) m_proxy_norm = std::sqrt(...);  
    return m_proxy_norm;  
}
```

# Opérateurs comme méthodes de classe

## Les opérateurs

- Possibilité de définir un opérateur comme méthode de classe;
- Pour opérateur unaire, pas d'arguments. Appliqué à l'objet appelant;
- Pour opérateur binaire, l'objet appelant se trouve à gauche de l'opérateur.

```
class Vecteur { ...  
    Vecteur operator + ( const Vecteur& v ) const; // <--- u+v  
    Vecteur operator - () const; // <--- -u  
    ...  
};  
  
Vecteur Vecteur::operator + ( const Vecteur& v ) const { ... }  
Vecteur Vecteur::operator -() const { ... }  
  
Vecteur w = u + v; // <--- idem que w = u.operator + ( v );  
Vecteur z = -u; // <--- idem que z = u.operator - ();
```

# Opérateurs de copie/déplacement

## Opérateur de copie

- Permet de copier les données d'un objet dans un objet existant;
- À ne pas confondre avec le constructeur de copie;
- Défini par défaut si non déclaré;
- On peut le définir par défaut ou le supprimer ( `default`, `delete` );
- Utilisé implicitement pour le retour par valeur dans une fonction d'une variable globale.

## Opérateur de déplacement

- Permet de déplacer les données d'un objet dans un objet existant;
- À ne pas confondre avec le constructeur de déplacement;
- Défini par défaut si non déclaré;
- On peut le définir par défaut ou le supprimer ( `default`, `delete` );
- Utilisé implicitement pour le retour par valeur dans une fonction d'une variable locale.

# Opérateurs de copie/déplacement

```
class Vecteur { ...  
    Vecteur& operator = ( const Vecteur& u ); // <--- Déclaration opérateur de copie  
    Vecteur& operator = ( Vecteur&& u ) = delete; // <--- Déclaration opérateur de déplacement supprimé  
    ...  
};  
  
Vecteur& Vecteur::operator = ( const Vecteur& u ) {  
    if ( this != &u ) { // <--- Si on cherche à copier un objet sur lui-même, on ne fait rien  
        ...  
    }  
    return *this; // On retourne l'objet appelant en référence  
}  
// On aurait eu un truc semblable pour l'opérateur de déplacement.
```



# Constructeur/Opérateur de copie

## Utilité du constructeur/opérateur de copie

- Permet de copier explicitement un objet sans le détruire :

```
Vecteur u( v ); // <--- Appel explicite au constructeur de copie
Vecteur w;
w = v; // <--- Appel explicite à l'opérateur de copie
```

- Mais aussi des appels implicites à l'opérateur ou au constructeur :
  - On appelle une fonction dont un argument est passé par valeur;
  - Quand on retourne par valeur un objet non local à une fonction.

```
Vecteur f( Vecteur u, Vecteur& w ) {
    w += u;
    return w;
}

Vecteur w = f(u,v); // Un appel au constructeur de copie et un à l'opérateur de copie
                    // L'appel au constructeur pour u passé par valeur
                    // L'appel à l'opérateur pour recopier le résultat dans w ( copie de v modifiée ).
```

# Constructeur/Opérateur de déplacement

## Utilité du constructeur/opérateur de déplacement

Permet de "voler" les données d'un objet qui ne sera plus utilisé/utilisable par la suite.

- Permet de déplacer les données d'un objet vers un autre explicitement :

```
Vecteur v(std::move(u)); // <--- Déplace les données de u dans v via le constructeur  
Vecteur w; w = std::move(u); // <--- Déplace les données de u dans w via l'opérateur
```

- Mais aussi des appels implicite lorsqu'une fonction retourne un objet local par valeur

```
std::vector<double> generate_coefs( int seed ) {  
    std::vector<double> w;  
    ...  
    return w;  
}  
  
std::vector<double> u = generate_coefs(24); // <--- Effectue un déplacement du vecteur  
                                           //         renvoyé dans le vecteur u
```

# Opérateurs de flux

## Problème

- Permet de lire/écrire un objet dans un flux d'entrée/sortie ( écran, fichier, etc. );

```
Vecteur u; ...  
std::cout << u << std::endl;  
std::ofstream fich("Sauvegarde.dat");  
fich << u;      fich.close();  
std::ifstream fich2("Sauvegarde.dat");  
fich2 >> u;     fich2.close();
```

- Objet à droite des opérateurs << et >> !
- Opérateurs doivent parfois lire/écrire dans partie privée de la classe

## Solutions

- Déclarer amis de la classe ([friend](#)) les opérateurs de flux : pose problème pour l'héritage...;
- Déclarer et définir des constructeurs/méthodes pour lire/écrire dans un flux;
- Opérateurs de flux définis via ces constructeurs/méthodes.

```
class Vecteur { ...  
    Vecteur( std::istream& inp );  
    std::ostream& print( std::ostream& out ) const;  
    ... };  
  
inline std::ostream& operator << ( std::ostream& out, const Vecteur& u ) { return u.print(out); }  
inline std::istream& operator >> ( std::istream& inp, Vecteur& u )  
{ u = Vecteur(inp); return inp; }
```

# Les accesseurs/modifieurs

## Accesseurs/modifieurs

- permettent d'accéder/modifier des données de l'objet;
- En général, doit être défini deux fois : en lecture seule et en lecture/écriture;
- L'opérateur [] permet d'accéder à un élément
- L'opérateur () peut aussi servir d'accession à un élément avec un indice multiple;
- Les itérateurs pour accéder aux éléments séquentiellement;

```
class Vecteur {
    typedef double* iterator;
    typedef const double* const_iterator;
    ...
    double& operator [] ( std::size_t i ) {
        assert(i < 3);
        return m_valeurs[i];
    }
    const double& operator [] ( std::size_t i ) const {
        assert(i < 3);
        return m_valeurs[i];
    }
    iterator begin() { return m_valeurs.data(); }
    const_iterator begin() const {
        return m_valeurs.data(); }
    iterator end() {
        return m_valeurs.data() + 3;
    }
    const_iterator end() const {
        return m_valeurs.data() + 3;
    }
};
```

## Usage

```
Vecteur a{{1.,3.,5.}};
a[2] = 4.;
double c = a[1];
// Utilisation des itérateurs
for ( auto& c : a )
    c += 1.5;
for ( auto iter = a.begin();
      iter != a.end(); ++iter )
    *iter = *iter / 2;
```

# Cas moins trivial

```

class Matrix {
    class iterator {
        iterator( Matrix& mat, int i, int j) :
            m_ref_mat(mat),
            m_pt_coef(mat.data()+i*mat.nbColumns()+j),
            m_irow(i), m_jcol(j) {}
        iterator( const iterator& it ) = default;
        iterator( iterator&& it ) = default;
        ~iterator() = default;

        bool operator != ( const iterator& it ) const {
            return m_pt_coef != it.m_pt_coef;
        }
        iterator& operator ++() {
            ++m_pt_coef;
            ++m_jcol;
            if (m_jcol > mat.nbColumns()) {
                m_jcol = 0;
                ++m_irow;
            }
        }
        double& operator * () const {
            return *m_pt_coef;
        }
    }
    class const_iterator { // Programmation similaire à iterator*
        ...
    }
    Matrix(...);
    ...
    iterator& begin() { return iterator(*this, 0, 0); }
    iterator& end  () { return iterator(*this, nbRows(), 0); }
    ...
};

```

# Les opérateurs de conversion

## Convertir un objet en une instance de sa classe

- Simplement déclarer et définir un constructeur prenant le type d'objet à convertir en argument

```
class Algebra::Vecteur {  
public: ...  
    // Permet de convertir un tableau dynamique en Vecteur algébrique  
    Vecteur( const std::vector<double>& arr ) : ... { ... }  
    ... };  
  
std::vector<double> arr; ...  
Algebra::Vecteur u(arr); // Convertit arr en vecteur algébrique
```

- La conversion implique des copies ! ( mais difficile de faire autrement ... )

## Convertir une instance de sa classe en un autre type d'objet

Supposons qu'on veut convertir une instance de type A en une instance de type B

- Soit la classe B est une classe conçue par le programmeur : on revient au cas plus haut en intervertissant A et B;
- La classe B est une classe d'une librairie extérieure : impossible de modifier la classe B !
- Utiliser les opérateurs de conversion : `operator Type()` en méthode de classe où Type est le nom du type dans lequel on veut convertir l'objet;
- Permet aussi de supprimer des interdépendances.

# Opérateur de conversion

## Conversion matrice pleine et matrice creuse

```
// SparseMatrix.hpp
class SparseMatrix { ... }
// PlainMatrix.hpp
# include "SparseMatrix.hpp"
class PlainMatrix {
    PlainMatrix( const SparseMatrix& spMat );// Convertit une matrice creuse en une matrice pleine
    ...
    operator SparseMatrix(); // Convertit une matrice pleine en matrice creuse.
};
...
```

- Si on avait défini deux constructeurs de conversion : un pour PlainMatrix, l'autre pour SparseMatrix, on aurait introduit une interdépendance, difficile à gérer à la production;
- Ici, SparseMatrix n'a pas connaissance de l'existence de PlainMatrix;

## Conversion Vecteur à tableau dynamique

```
namespace Algebra {
    class Vecteur { ...

    explicit operator std::vector<double>() const; // Conversion en tableau dynamique.
    ...
};
```

- Ici, pas le choix, on ne peut modifier la librairie vector, donc uniquement un opérateur de conversion possible.

# Attributs et méthodes de classe

## Attribut de classe

- Donnée associée à une classe et non à des instances de la classe;
- Peut-être accéder sans instance de classe

```
namespace Parallel {  
class Context {  
public: ...  
    static Communicator global_com;  
    ...  
};  
Communicator Context::global_com{};
```

## Méthode de classe

- Méthode associée à la classe et non pas à une instance de classe.

```
class Context { ...  
public:  
    static std::shared_ptr<Context> get() {  
        if (glob_ptr == nullptr) glob_ptr = std::make_shared<Context>();  
        return glob_ptr;  
    }  
private:  
    Context() { ... };  
    static std::shared_ptr<Context> glob_ptr;  
};
```



# Exercice : Nuage de points ( suite )

## Rajout d'opérateurs

- ➊ Rajouter les opérateurs de copie/déplacement
- ➋ Accéder au  $i^{\text{e}}$  point  $p_i$  pour le lire/modifier;
- ➌ Rajouter les opérateurs adéquats pour que le test suivant fonctionne :

```
CloudOfPoints cop1, cop2;
...
CloudOfPoints cop3 = cop1 + cop2; // cop3 = fusion de cop1 et cop2
Point tr{1.,0.,0.};
cop3 += tr; // Translation des points par le vecteur tr
// Affiche nombre de points et les 1ers et derniers points...
std::cout << "cop3 : " << std::string(cop3) << std::endl;
// Sauvegarde le nuage de points :
ofstream fich("cloud.dat"); fich << cop3; fich.close();
```

## Rajout de méthodes

- ➊ Rajouter une méthode donnant le nombre de points,
- ➋ Pouvoir itérer sur les points du nuage;
- ➌ Calculer le point barycentre du nuage de point.

# L'héritage

## Quand utiliser l'héritage

- Traduit une relation "être" : une matrice symétrique **est** une matrice carrée;
- Sert lorsqu'on veut factoriser des services communs à plusieurs objets;
- Sert pour spécialiser un objet;
- Ne pas confondre avec l'aggrégation;
- Le choix de hiérarchie des classes est guidée par l'application voulue.

## Exemple de spécialisation : Vecteur algébrique

```
namespace Algebra {  
    class Vecteur : public std::vector<double>  
    {  
        // Vecteur a accès à tous les services d'un std::vector  
        // + d'autres services fournis dans la classe  
        Vecteur();  
        Vecteur( std::size_t dim );  
        ...  
        // Spécialisation en vecteur algébrique  
        Vecteur operator + ( const Vecteur& u ) const; // Addition de deux vecteurs...  
        ...  
    };  
}
```

# Héritage par factorisation

## Héritage matrices

```
class Quadrilatere {
public:
    ...
    double calculAire() const { return 0; }
};
class Rectangle : public Quadrilatere {
    ...
    double calculAire() const
    { return m_length * m_height; }
private:
    double m_length, m_height;
};
class Square : public Quadrilatere {
    ...
    double calculAire() const {
        return m_length * m_length;
    }
};
```

- On ne sait pas calculer l'aire d'un quadrilatère quelconque;
- Ça ne marche pas :

```
double addAire( Quadrilatere a,
                Quadrilatere b ) {
    return a.calculAire()+b.calculAire();
}
int main() {
    Square c(3.0);
    Rectangle r(2.0,3.0);
    std::cout << c.calculAire() << "+" <<
               << r.calculAire() << "=" <<
               << addAire(c,r) << std::endl;
    return EXIT_SUCCESS;
}
```

sort à l'exécution :

```
9 + 6 = 0
```

# Pourquoi ça ne marche pas ?

## Passage des arguments par valeur

- On copie les objets passés en argument en tant que `Quadrilatere`;
- Le véritable type de l'objet passé en paramètre est oublié;
- Dans la fonction on ne manipule plus que des copies de ces objets en tant que `Quadrilatere`.
- **Solution** : Passer les arguments par références sur des objets constants.

## Appel "statique" à une méthode

- En C++, pour raison d'efficacité, le compilateur essaie de décider quelle fonction appelée à la compilation;
- C'est le cas pour toutes les méthodes déclarées "normalement";
- Ici, il voit qu'on manipule des objet de type `Quadrilatere`, il décide donc à la compilation d'appeler la méthode `calculAire` de la classe `Quadrilatere`
- **Solution** : Pour que la méthode appelée soit choisie dynamiquement en fonction des objets passés, il faut précéder la déclaration de la méthode du mot clef `virtual`;
- Une méthode virtuelle redéfinie dans une classe fille doit posséder la même signature que la fonction virtuelle de base;
- Seul le type de retour peut changer à condition qu'il soit un pointeur ou une référence et que l'objet renvoyé hérite de l'objet renvoyé dans la classe mère;
- Dans beaucoup d'autres langages, les fonctions sont par défaut virtuelles ( Python, Java... ).

# Héritage par factorisation (suite)

## Héritage matrices

```
class Quadrilatere {
public:
    ...
    virtual double calculAire() const { return 0; }
};
class Rectangle : public Quadrilatere {
    ...
    virtual double calculAire() const override
    { return m_length * m_height; }
private:
    double m_length, m_height;
};
class Square : public Quadrilatere {
    ...
    virtual double calculAire() const override {
        return m_length * m_length;
    }
};
```

- Le mot clef `override` permet de s'assurer qu'on redéfinit bien une méthode virtuelle de la classe de base;
- Ça marche ! :

```
double addAire( const Quadrilatere& a,
               const Quadrilatere& b ) {
    return a.calculAire()+b.calculAire();
}
int main() {
    Square c(3.0);
    Rectangle r(2.0,3.0);
    std::cout << c.calculAire() << "+" <<
               << r.calculAire() << "=" <<
               << addAire(c,r) << std::endl;
    return EXIT_SUCCESS;
}
```

sort à l'exécution :

```
9 + 6 = 15
```

# Fonctions virtuelles pures

## Problématique

- 1. Que mettre dans la fonction `calculAire` de `Quadrilatere` ?
- 2. En fait, ne connaissant pas à ce niveau le type de quadrilatère, on ne sait pas calculer l'aire;
- 3. On définit une **fonction virtuelle pure** : la fonction n'est que déclarée et on devra la surcharger dans les classes dont on instanciera effectivement des objets;
- 4. La classe `Quadrilatere` devient une **classe abstraite** : on ne peut instancier directement des objets de cette classe;
- 5. Toute classe héritant directement ou indirectement ( au travers une hiérarchie d'héritage ) d'une classe abstraite devient *concrète* que si toutes les méthodes virtuelles pures ont été définies dans la classe ou dans la hiérarchie de classe dont elle hérite.

```
class Quadrilatere {  
    ...  
    // Déclaration méthode virtuelle pure  
    virtual double calculAire() const = 0;  
};  
...  
int main()  
{  
    Quadrilatere q; // Erreur à la compilation !  
    return 0;  
}
```

# Conception par contrat

## Carré/Rectangle : une autre hiérarchie de classe ?

- Pourquoi ne pas avoir spécialisé la classe `Rectangle` en classe `Carré` ?
- Le rectangle doit stocker sa longueur et sa hauteur, le carré que sa longueur;
- Méthodes pour changer la longueur/hauteur du rectangle : un carré ne vérifiera plus l'égalité de la hauteur et de la longueur en poscondition de ces méthodes...
- Alors pourquoi ne pas dire qu'un rectangle hérite du carré ?
- Conception peut naturelle mais qui supprime le problème des attributs;
- On peut passer un rectangle en tant que carré dans des fonctions dont une pré-condition d'entrée est que la longueur soit égale à la hauteur.
- Notre précédente hiérarchie de classe ne posait pas ces problèmes...

## Conception par contrat

- Méthode proposée par Bertrand Meyer pour son langage Eiffel;
- **Pré-condition sur les objets** : une condition que doit vérifier les données d'un objet à l'entrée d'une méthode/fonction;
- **Post-condition sur les objets** : une condition que doit vérifier les données d'un objet à la sortie d'une fonction.
- **Règle pour s'assurer d'avoir un héritage cohérent** :
  - Les pré-conditions sur les objets des méthodes de la classe héritée doivent contenir les préconditions sur les objets de la classe fille;
  - Les post-conditions sur les objets des méthodes de la classe héritée doivent être contenues dans les pré-conditions des méthodes de la classe fille;

# Le destructeur

## Destructeur non virtuel

```
class A { ...
    ~A() { std::cout << "~A" << std::endl; }
... };
class B : public A { ...
    B(int n) : m_coefs(new double[n]) {}
    ~B() { delete [] m_coefs; std::cout << "~B" << s
        double* m_coefs; ... };
int main() {
    A* pt_a = new B(10);
    delete pt_a;
    return 0; }
```

### Affichage à l'exécution

~A

- Seul le destructeur de A est appelé;
- Le tableau m\_coefs n'est jamais détruit.
- L'appel au destructeur est statique, en fonction du type du pointeur, et non de l'objet.

## Destructeur virtuel

```
class A { ...
    virtual ~A() { std::cout << "~A" << std::endl; }
... };
class B : public A { ...
    B(int n) : m_coefs(new double[n]) {}
    ~B() { delete [] m_coefs; std::cout << "~B" << s
        double* m_coefs; ... };
int main() {
    A* pt_a = new B(10);
    delete pt_a;
    return 0; }
```

### Affichage à l'exécution

~B

~A

- On appelle le bon destructeur ( qui appelle automatiquement celui de A );
- L'appel au destructeur est dynamique, il dépend du type de l'objet pointé.



# Héritage et opérateurs

## Opérateur de copie

```
class Quadrilatere{
    ....
    virtual Quadrilatere& operator = ( const Quadrilatere& ) = 0; };
class Carre : public Quadrilatere {
    virtual Carre& operator = ( const Quadrilatere& r ); };
```

- ➊ Plusieurs problèmes :

- ➋ Dans l'argument de copie de la classe Carre on peut passer un rectangle ou tout autre objet héritant de Quadrilater...
- ➌ En détectant le type d'objet passé en argument, on va très vite arriver à une combinaison exponentielle.

- ➍ **Solution** : Ne pas utiliser l'opérateur de copie mais une méthode clone/copy comme en Java ou en Python.

## Opérateurs arithmétiques

- ➊ Impossible de les définir dans la classe de base si ils doivent renvoyer une valeur du type de la classe de base. Il faut passer par des méthodes prenant en argument les deux arguments d'entrée et la sortie.
- ➋ On pourra alors définir les opérateurs arithmétiques dans les classes concrètes;
- ➌ Attention cependant au nombre de combinaison...

```
class Matrice {
    ...
    virtual Vecteur operator * ( const Vecteur& u ) const = 0; // OK
    Matrice operator + ( const Matrice& mat ) const = 0; // Impossible, Matrice est une classe abstraite
    virtual void add( const Matrice& mat, Matrice& resMat ) const = 0; // OK
    ... };
```

# Nuage de points ( suite et fin )

## Différents nuages

- On veut créer différentes sortes de nuages de points : un représentant des pressions, d'autres des champs de vitesse à des endroits mesurés par des capteurs et un nuage qui représente des particules physique ( de poussière par exemple ) qui seront convectés par le champs de vitesse...
- Pour chacun de ces nuages de points, on voudrait pouvoir calculer la pression, la vitesse ou la position moyenne du nuage;
- Convecter les particules à l'aide du champs de vitesse;
- Afficher sous forme de chaîne de caractère chaque nuage à l'écran avec ses valeurs associées.

# Bonus : les expressions littérales

```
namespace Units
{
    namespace Distance
    {
        class Metre
        {
        public:
            class DoubleIsM{}; // A Tag
            explicit constexpr Metre(DoubleIsM, double dist) : m_dist(dist) {}

            explicit operator double() const { return m_dist; }

            Metre operator + ( const Metre& m )
            {
                return Metre{DoubleIsM{}, m_dist + m.m_dist};
            }

            std::ostream& print( std::ostream& out ) const
            {
                out << m_dist << "m";
                return out;
            }

            template<typename U>
            Metre operator = ( const U& d ) {
                if ( this != &d ) {
                    Metre md = Metre(d);
                    m_dist = md.m_dist;
                }
                return *this;
            }

        private:
            double m_dist;
    }
}
```

# Bonus : les expressions littérales(suite)

```
};

inline std::ostream& operator << ( std::ostream& out, const Metre& m ) {
    return m.print(out);
}

class Kilometre
{
public:
    class DoubleIsKm{}; // A Tag
    explicit constexpr Kilometre(DoubleIsKm, double dist ) : m_dist(dist) {}
    Kilometre( const Metre& d ) : m_dist(0.001*double(d)) {}

    explicit operator double() const { return m_dist; }

    operator Metre()
    {
        return Metre{Metre::DoubleIsM{}, 1000.*double(m_dist)};
    }

    Kilometre operator + ( const Kilometre& m )
    {
        return Kilometre{DoubleIsKm{}, m_dist + m.m_dist};
    }

    std::ostream& print( std::ostream& out ) const
    {
        out << m_dist << "km";
        return out;
    }
}
```