PROGRAMMATION ORIENTÉE OBJET

STL - HÉRITAGE - POLYMORPHISME

charles.meunier@u-bourgogne.fr

STANDARD TEMPLATE LIBRARY

charles.meunier@u-bourgogne.fr

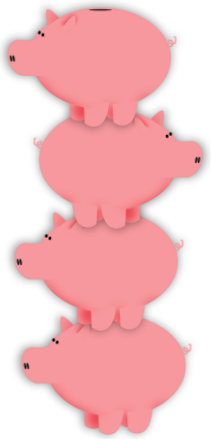
GÉNÉRICITÉ

Principe : créer un **MODÈLE** de classe, de fonction ou d'algorithme **RÉUTILISABLE POUR DIFFÉRENTS TYPES DE DONNÉES**.

Pile d'assiettes

47
19
20
96
8

Pile d'entiers



Pile de cochons

TOUJOURS LE MÊME PRINCIPE DE FONCTIONNEMENT

charles.meunier@u-bourgogne.fr

GÉNÉRICITÉ EN C++

Le C++ permet de créer des éléments génériques

Le type de donnée manipulé est

- **INCONNU** à la conception de l'élément
- **SPECIFIÉ** au moment de l'instanciation de la classe générique ou de l'appel de la fonction générique

EXEMPLE : LA PILE - DÉCLARATION

pile.h

```
template<class T> class Pile
{
    private:
        std::vector<T> elements;

    public:
        void empiler(const T& element);
        T depiler();
        int getTaille() const;
};

template<class T> void Pile<T>::empiler(const T& element)
{
    ...
}

template<class T> T Pile<T>::depiler()
{
    ...
}

template<class T> int getTaille() const
{
    ...
}
```

IMPORTANT

TOUT LE CODE DU TEMPLATE DOIT SE TROUVER DANS UN SEUL ET MÊME FICHIER

EXEMPLE : LA PILE - UTILISATION

main.cpp

```
#include "pile.h"
#include <string>

int main()
{
    Pile<int> pileEntiers;

    Pile<std::string> pileChainesCaracteres;

    pileEntiers.empiler(18);

    pileChainesCaracteres.empiler("toto");

    return 0;
}
```

STANDARD TEMPLATE LIBRARY

La STL du C++ repose sur le principe de généricité et propose tout un ensemble de modèles

- de conteneurs (list, vector, map, …)
- d'algorithmes (tri, recherche, insertion, …)
- de pointeurs (iterator, shared_ptr, unique_ptr)
- …

CONTENEURS DE LA STL

- array : conteneur de taille statique qui stocke les éléments de manière contiguë en mémoire
- vector : conteneur de taille variable qui stocke les éléments de manière contiguë en mémoire
- list : conteneur doublement chainé dont les éléments peuvent être ajoutés ou supprimés à n'importe quelle position, mais accès séquentiel (pas d'opérateur [ ])
- deque : mix entre vector et list, ajout rapide au début et à la fin, conserve l'adresse des éléments existants.
- set : conteneur qui stocke des éléments uniques, triés par ordre croissant. Les éléments ne peuvent être modifiés une fois insérés
- map : conteneur qui stocke les données sous forme de paire clé => valeur, triés par ordre croissant de la clé.
- stack : conteneur qui suit le principe LIFO (Last In First Out)
- queue : conteneur qui suit le principe FIFO (First In First Out)

EXEMPLE : VECTOR

```cpp
#include <vector>

int main()
{
    //Création d'un vector d'entiers
    std::vector<int> entiers

    //Ajout d'un élément à la fin du vector
    entiers.push_back(44);

    //Accès au 1er élément du vector
    int val = entiers.at(0);

    return 0;
}
```

EXEMPLE : PARCOURIR UN VECTOR

```
std::vector<int> entiers = { 10, 20, 30, 40 };

for(std::vector<int>::iterator it = entiers.begin(); it < entiers.end(); ++it)
{
    std::cout << *it << std::endl;
}
```


```
std::vector<int> entiers = { 10, 20, 30, 40 };

for(auto it = entiers.begin(); it < entiers.end(); ++it)
{
    std::cout << *it << std::endl;
}
```

ITERATOR

Les iterator sont utilisés pour manipuler les conteneurs de la STL

Ce sont des sortes de pointeurs qui pointent sur une valeur du conteneur et qui peuvent passer d'une valeur à l'autre en fonction du type de conteneur.

Il existe plusieurs types d'iterator :

- iterator : permet de parcourir les éléments du conteneur du début à la fin
- const_iterator : idem mais les valeurs du conteneur ne peuvent pas être modifiées
- reverse_iterator : permet de parcourir les éléments à partir de la fin du conteneur
- const_reverse_iterator : idem mais les valeurs du conteneur ne peuvent pas être modifiées.

EXEMPLE : SUPPRIMER UN ÉLÉMENT D'UN VECTOR

```
std::vector<int> entiers = { 10, 20, 30, 40 };

//Supprimer le 3ème élément du vector
entiers.erase(entiers.begin() + 2);
```

- **`begin()`** retourne un iterator sur le premier élément du vector
- **`end()`** retourne un iterator sur la position située **APRÈS LE DERNIER ÉLÉMENT**.

EXEMPLE : INSÉRER UN ÉLÉMENT DANS UN VECTOR

```
std::vector<int> entiers = { 10, 20, 30, 40 };

//Insère une valeur avant le quatrième élément
entiers.insert(entiers.begin() + 3, 35);
```

- **`begin()`** retourne un iterator sur le premier élément du vector
- **`end()`** retourne un iterator sur la position située **APRÈS LE DERNIER ÉLÉMENT**.

TEMPLATE ET TYPES DE DONNÉE VALIDES

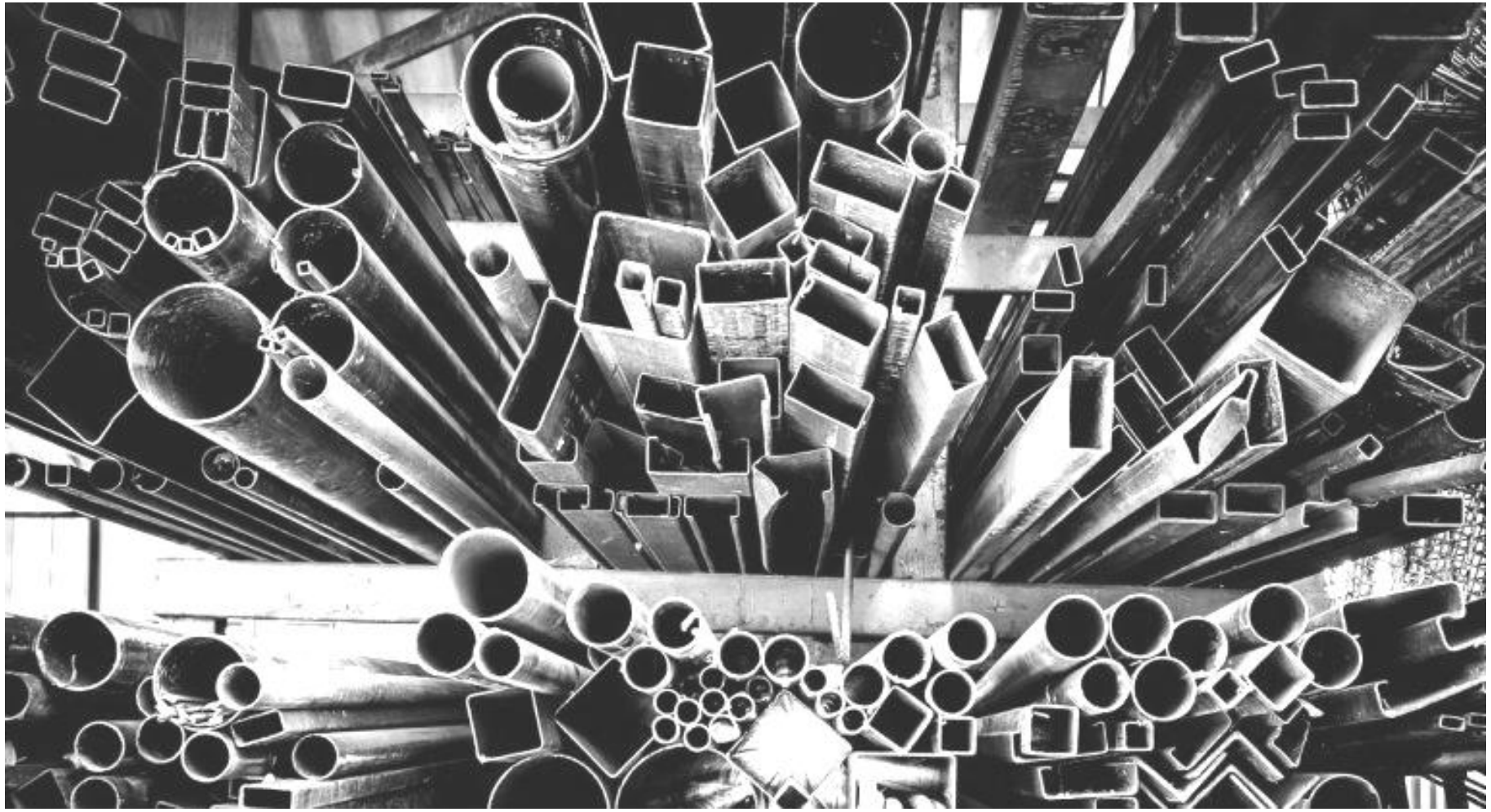
Générique ne veut pas dire que le modèle est valable pour tous les types de donnée. Cela dépend de l'usage qui en est fait au sein des traitements réalisés par le modèle.

Par exemple, un **`vector`** ne peut gérer que les types de donnée qui :

- peuvent être recopiés (types natifs ou classes disposant d'un constructeur par recopie)
- sont affectables (types natifs ou classes disposant de l'opérateur =)

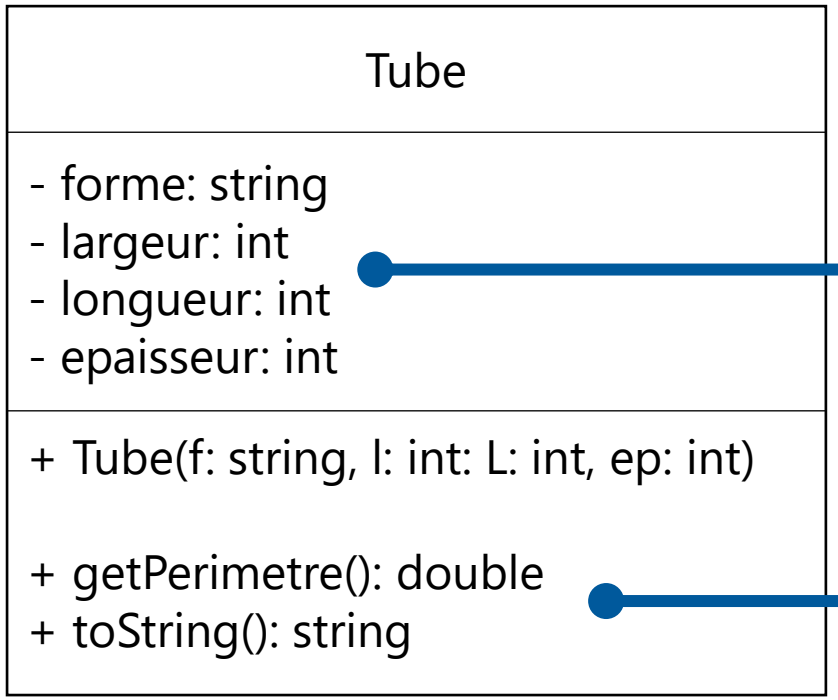
HÉRITAGE

charles.meunier@u-bourgogne.fr



charles.meunier@u-bourgogne.fr

PREMIÈRE MODÉLISATION



`largeur` et **`longueur`** sont deux attributs cohérents pour un tube rectangulaire. En revanche, pour un tube carré la longueur n'est pas nécessaire, et pour un tube rond, on parlerait plutôt de diamètre.

Les traitements à réaliser seront différents en fonction de la forme du tube. Il va falloir utiliser des if/then/else ou des switch. Le rôle de ces fonctions est trop large.

DEUXIÈME MODÉLISATION

TubeRect	TubeCarré	TubeRond
- largeur: int - longueur: int - epaisseur: int	- largeur: int - epaisseur: int	- diametre: int - epaisseur: int
+ TubeRect(l: int: L: int, ep: int) + getPerimetre(): double + toString(): string	+ TubeCarre(l: int:, ep: int) + getPerimetre(): double + toString(): string	+ TubeRond(d: int:, ep: int) + getPerimetre(): double + toString(): string

Les attributs de chaque classe sont cohérents avec le concept représenté. Chaque méthode a un traitement bien spécifique à réaliser.

COMMENT GÉRER UNE LISTE DE TUBES DANS L'APPLICATION ?

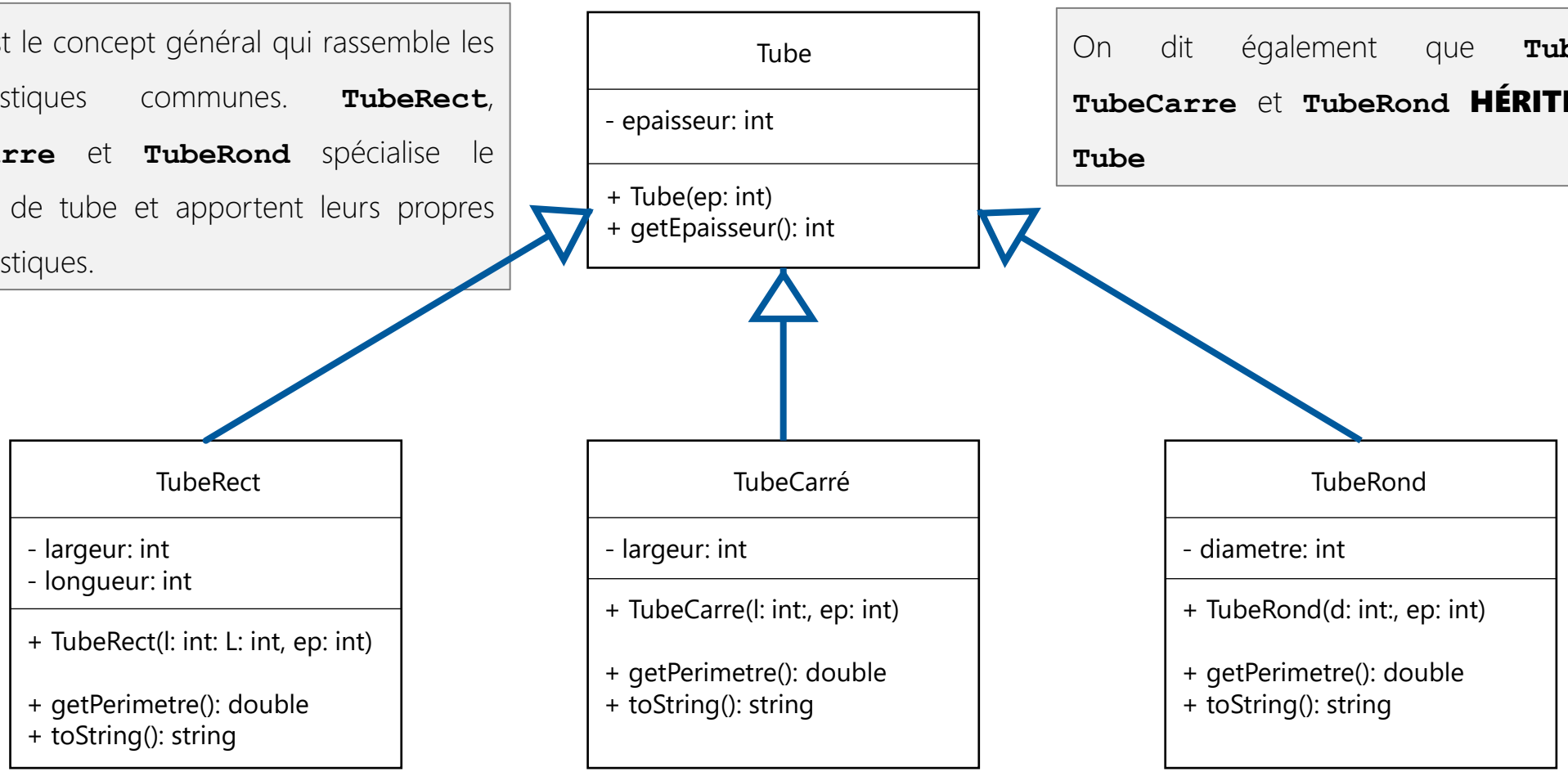
En C++, un conteneur (tableau, vector, ...) ne peut contenir que des objets du même type. Il faudra une liste par type de tubes, soit trois vector dans le cas présent :

```
int main()
{
    std::vector<TubeRect> tubesRectangulaires;
    std::vector<TubeCarre> tubesCarres;
    std::vector<TubeRond> tubesRonds;

    ...

    return 0;
}
```

Si on ajoute de nouveaux types de tubes, il faut créer de nouveaux vectors et modifier toutes les sections de l'application où ces vectors interviennent.

GÉNÉRALISATION / SPÉCIALISATION

Tube est le concept général qui rassemble les caractéristiques communes. **TubeRect**, **TubeCarre** et **TubeRond** spécialise le concept de tube et apportent leurs propres caractéristiques.



On dit également que **TubeRect**, **TubeCarre** et **TubeRond** **HÉRITENT** de **Tube**

SYNTAXE DE LA CLASSE TUBE EN C++

Pas de syntaxe particulière pour la classe mère.

tube.h

```
class Tube
{
    private:
        int epaisseur;

    public:
        Tube(int ep);

        int getEpaisseur();
}
```

tube.cpp

```
#include "tube.h"

Tube::Tube(int ep)
    : epaisseur(ep)
{
}

int Tube::getEpaisseur()
{
    return epaisseur;
}
```

SYNTAXE DE LA CLASSE TUBERECT EN C++

tube-rect.h

```
#include "tube.h"
#include <string>

class TubeRect: public Tube
{
    private:
        int largeur;
        int longueur;

    public:
        TubeRect(int ep);
        double getPerimetre();
        std::string toString();
}
```

L'attribut **epaisseur** de la classe **Tube** est privé et n'est donc pas accessible directement depuis la classe fille.

tube-rect.cpp

```
#include "tube-rect.h"
#include <sstream>

TubeRect::TubeRect(int l, int L, int ep)
    : Tube(ep), largeur(l), longueur(L)
{
}

double TubeRect::getPerimetre()
{
    return 2 * (largeur + longueur);
}

std::string TubeRect::toString()
{
    std::stringstream stream;
    stream << "Tube Rectangulaire - " << largeur;
    stream << " x " << longueur << " x " << getEpaisseur();

    return stream.str();
}
```

charles.meunier@u-bourgogne.fr

SYNTAXE DE LA CLASSE TUBECARRE EN C++

tube-carre.h

```
#include "tube.h"
#include <string>

class TubeCarre: public Tube
{
    private:
        int largeur;

    public:
        TubeCarre(int l, int ep);
        double getPerimetre();
        std::string toString();
}
```

tube-carre.cpp

```
#include "tube-rect.h"
#include <sstream>

TubeCarre::TubeCarre(int l, int ep)
    : Tube(ep), largeur(l)
{
}

double TubeCarre::getPerimetre()
{
    return 4 * largeur;
}

std::string TubeCarre::toString()
{
    std::stringstream stream;
    stream << "Tube Carre - " << largeur;
    stream << " x " << largeur << " x " << getEpaisseur();

    return stream.str();
}
```

SYNTAXE DE LA CLASSE TUBEROND EN C++

tube-rond.h

```
#include "tube.h"
#include <string>

class TubeRond: public Tube
{
    private:
        int diametre;

    public:
        TubeRond(int d, int ep);
        double getPerimetre();
        std::string toString();
}
```

tube-rond.cpp

```
#include "tube-rect.h"
#include <sstream>
#include <numbers>

TubeRond::TubeRond(int d, int ep)
    : Tube(ep), diametre(d)
{
}

double TubeRond::getPerimetre()
{
    return std::numbers::pi * diametre;
}

std::string TubeRond::toString()
{
    std::stringstream stream;
    stream << "Tube Rond - Ø" << diametre;
    stream << " x " << getEpaisseur();

    return stream.str();
}
```

charles.meunier@u-bourgogne.fr

LISTE DE TUBES DANS MAIN

main.cpp

```
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube> tubes;

    tubes.push_back(TubeRect(50, 100, 3));
    tubes.push_back(TubeCarre(30, 1));
    tubes.push_back(TubeRond(40, 2));

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

! ERREUR !

LA CLASSE TUBE NE DISPOSE PAS D'UNE MÉTHODE `toString()`

AJOUT DE LA FONCTION TOSTRING À LA CLASSE TUBE

tube.h

```
class Tube
{
    private:
        int epaisseur;

    public:
        Tube(int ep);

        int getEpaisseur();

        std::string toString();
}
```

tube.cpp

```
...

std::string Tube::toString()
{
    return "Un tube !";
}
```

LISTE DE TUBES DANS MAIN

main.cpp

```cpp
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube> tubes;

    tubes.push_back(TubeRect(50, 100, 3));
    tubes.push_back(TubeCarre(30, 1));
    tubes.push_back(TubeRond(40, 2));

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

CONSOLE

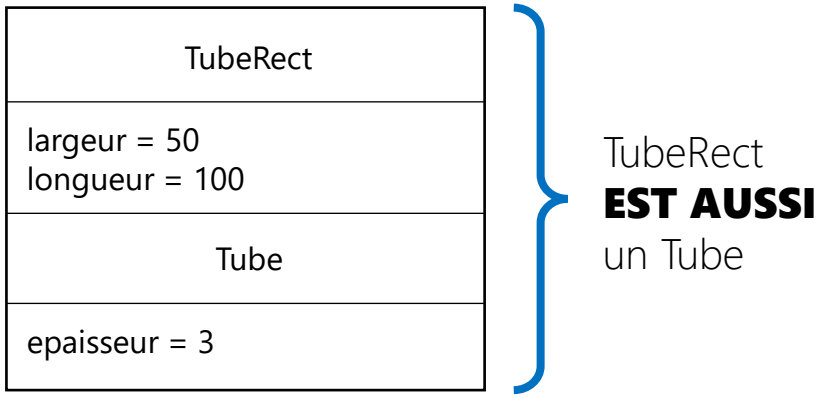
```
Un tube !
Un tube !
Un tube !
```

Pourquoi est-ce la méthode **`toString()`** de **`Tube`** qui est appelée et non celle des classes filles ?

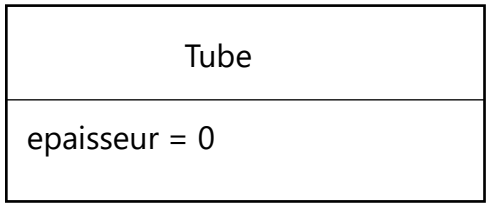
EN MÉMOIRE...

```
tubes.push_back(TubeRect(50, 100, 2));
```

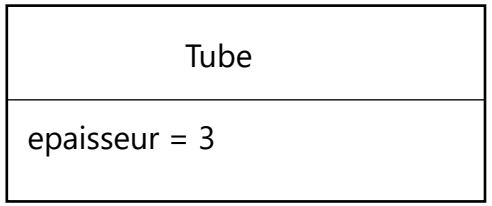
1. Création d'une instance de TubeRect



2. Ajout d'une nouvelle instance de Tube au vector



3. Recopie de l'instance de TubeRect dans l'instance de Tube du vector



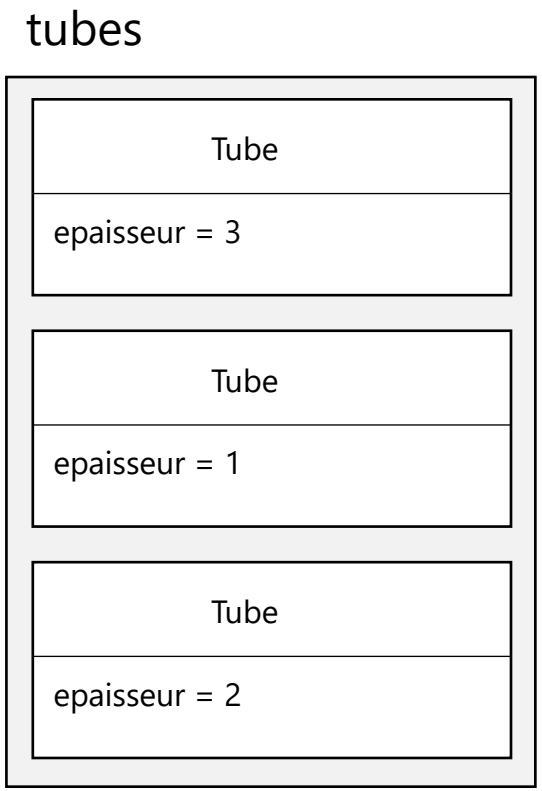
EN MÉMOIRE...

À l'issue de ce code...

```
tubes.push_back(TubeRect(50, 100, 3));
tubes.push_back(TubeCarre(30, 1));
tubes.push_back(TubeRond(40, 2));
```

... **LE VECTOR NE CONTIENT QUE DES INSTANCES DE TUBE :**

C'est donc la méthode **`toString()`** de la classe Tube qui est appelée systématiquement.



VERS UNE LISTE DE POINTEURS SUR TUBES

main.cpp

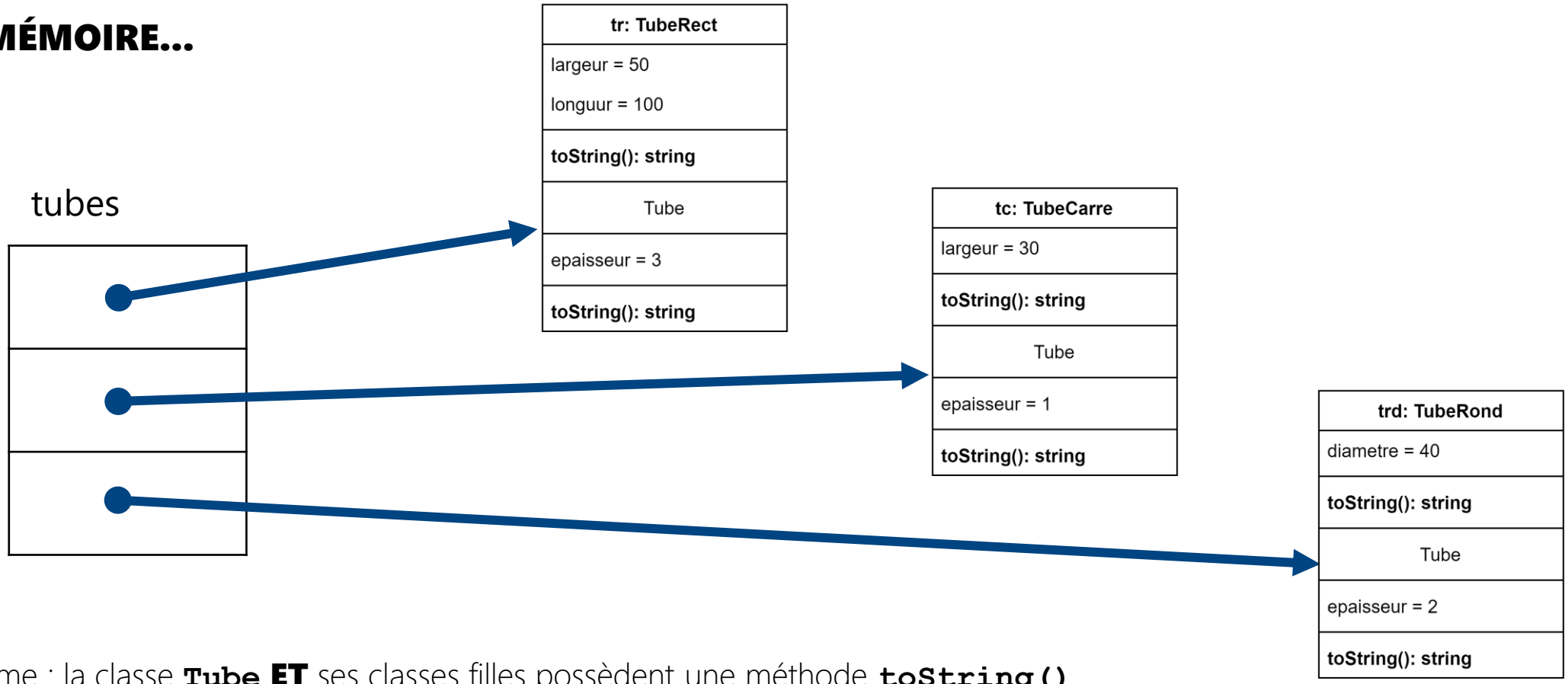
```cpp
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube*> tubes;

    TubeRect tr(50, 100, 3);
    TubeCarre tc(30, 1);
    TubeRond trd(40, 2);

    tubes.push_back(&tr);
    tubes.push_back(&tc);
    tubes.push_back(&trd);

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it)->toString() << std::endl;
    }

    return 0;
}
```

CONSOLE

```
Un tube !
Un tube !
Un tube !
```

Pourquoi est-ce la méthode **`toString()`** de **`Tube`** qui est appelée et non celle des classes filles ?

EN MÉMOIRE...



Problème : la classe **`Tube`** **ET** ses classes filles possèdent une méthode **`toString()`**.

`tubes` étant un **`vector`** de **`Tube`**, le compilateur privilégie la méthode **`toString()`** de **`Tube`**.

VIRTUAL

Le mot clé **virtual** indique que les classes filles peuvent **REDÉFINIR** la méthode de la classe mère. Ainsi, lorsque le compilateur se trouve dans la situation précédente, il privilégie la méthode redéfinie par la classe fille au lieu de celle de la classe mère. C'est ce qu'on appelle le **POLYMORPHISME**.

tube.h

```
class Tube
{
    private:
        int epaisseur;

    public:
        Tube(int ep);

        int getEpaisseur();

        virtual std::string toString();
}
```

tube.cpp

```
...

std::string Tube::toString()
{
    return "Un tube !";
}
```

Le mot clé **virtual** n'est utilisé que dans le fichier .h

RÉSULTAT AVEC LE MOT CLÉ VIRTUAL

main.cpp

```cpp
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube*> tubes;

    TubeRect tr(50, 100, 3);
    TubeCarre tc(30, 1);
    TubeRond trd(40, 2);

    tubes.push_back(&tr);
    tubes.push_back(&tc);
    tubes.push_back(&trd);

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

CONSOLE

```
Un tube rectangulaire 50 x 100 x 3
Un tube carre 30 x 1
Un tube rond Ø40 x 2
```

TOUTEFOIS...

main.cpp

```
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube*> tubes;

    TubeRect tr(50, 100, 3);
    TubeCarre tc(30, 1);
    TubeRond trd(40, 2);

    tubes.push_back(&tr);
    tubes.push_back(&tc);
    tubes.push_back(&trd);

    Tube t;
    tubs.push_back(&t);

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

CONSOLE

```
Un tube rectangulaire 50 x 100 x 3
Un tube carre 30 x 1
Un tube rond Ø40 x 2
Un tube !
```

Cela a-t-il du sens de pouvoir créer un simple **`Tube`** ?

Que doit afficher la fonction **`toString()`** d'un **`Tube`** ?

Que doit retourner la méthode **`getPerimetre()`** d'un **`Tube`** ?

MÉTHODE ABSTRAITE

tube.h

```
class Tube
{
    private:
        int epaisseur;

    public:
        Tube(int ep);

        int getEpaisseur();

        virtual std::string toString() = 0;
        virtual double getPerimetre() = 0;
}
```

- Une méthode abstraite est une **MÉTHODE VIRTUELLE DONT LE CODE N'EST PAS DÉFINI**.
- Il n'existe pas de code pour ces méthodes ni dans le fichier tube.h, ni dans le fichier tube.cpp, en dehors des signatures.
- **UNE CLASSE CONTENANT UNE MÉTHODE ABSTRAITE EST ELLE-MÊME ABSTRAITE**.
- Une classe abstraite **NE PEUT PAS ÊTRE INSTANCIÉE**.
- Toute classe qui hérite d'une classe abstraite est elle même abstraite, sauf si elle définit le code de toutes les méthodes abstraites de son ou ses parents.

TUBE EST À PRÉSENT UNE CLASSE ABSTRAITE

main.cpp

```
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube*> tubes;

    TubeRect tr(50, 100, 3);
    TubeCarre tc(30, 1);
    TubeRond trd(40, 2);

    tubes.push_back(&tr);
    tubes.push_back(&tc);
    tubes.push_back(&trd);

    Tube t;
    tubs.push_back(&t);

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

! ERREUR !

TUBE EST UNE CLASSE ABSTRAITE.

ON NE PEUT DONC PAS CRÉER D'INSTANCE DE TYPE TUBE.

CODE FINAL ?

main.cpp

```cpp
#include<vector>
#include<iostream>

int main()
{
    std::vector<Tube*> tubes;

    TubeRect tr(50, 100, 3);
    TubeCarre tc(30, 1);
    TubeRond trd(40, 2);

    tubes.push_back(&tr);
    tubes.push_back(&tc);
    tubes.push_back(&trd);

    for(auto it = tubes.begin(); it < tubes.end(); ++it)
    {
        std::cout << (*it).toString() << std::endl;
    }

    return 0;
}
```

CONSOLE

```
Un tube rectangulaire 50 x 100 x 3
Un tube carre 30 x 1
Un tube rond Ø40 x 2
```

Tout fonctionne comme attendu.

La classe **`Tube`** ne définit plus de code pour les méthodes **`toString`** et **`getPerimetre`**, ce qui n'avais pas de sens.

Le rôle de chaque élément est bien défini.

MAIS...

OVERRIDE - EXEMPLE

mere.h

```
class Mere
{
    public:
        Mere();

        virtual void methode() const;
}
```

fille.h

```
class Fille: public Mere
{
    public:
        Fille();

        void methode();
}
```

LES SIGNATURES SONT DIFFÉRENTES. FILLE NE REDÉFINIT PAS LA MÉTHODE DE MERE.

Mais ça ne pose pas de problème au compilateur.

Pour lui, Fille déclare une nouvelle méthode sans chercher à redéfinir celle de sa classe mère

OVERRIDE

Que se passe-t-il si au cours de la vie du projet, la signature d'une méthode d'une classe mère évolue ?

Dans le cas d'une méthode abstraite, le compilateur va indiquer que vous ne pouvez plus instancier les classes filles car, ne redéfinissant pas cette nouvelle méthode abstraite, elles sont elles-mêmes devenues abstraites.

Mais pour ce qui est des méthodes "classiques", le compilateur ne signalera rien. La classe mère a ses méthodes, les classes filles les leurs, mais il n'y a plus de redéfinition de fonction.

Pour éviter ce problème, C++11 intègre le mot clé **`override`** qui permet aux classes filles d'indiquer explicitement au compilateur qu'elles sont censées redéfinir une méthode de leur classe mère. Si aucune signature ne correspond dans la classe mère, une erreur de compilation se produira.

OVERRIDE - EXEMPLE

mere.h

```
class Mere
{
    public:
        Mere();

        virtual void methode() const;
}
```

fille.h

```
class Fille: public Mere
{
    public:
        Fille();

        void methode() override;
}
```

LES SIGNATURES SONT DIFFÉRENTES. FILLE NE REDÉFINIT PAS LA MÉTHODE DE MERE ALORS QU'ELLE INDIQUE LE FAIRE.

Une erreur sera levée à la compilation.

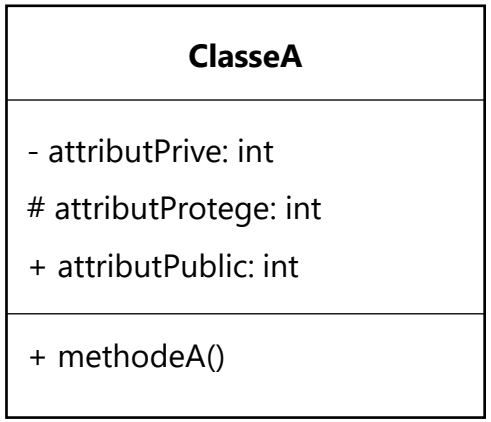
UTILISEZ SYSTÉMATIQUEMENT LE MOT CLÉ OVERRIDE LORSQUE VOUS REDÉFINISSEZ UNE FONCTION.

charles.meunier@u-bourgogne.fr

MODIFICATEURS D'ACCÈS

charles.meunier@u-bourgogne.fr

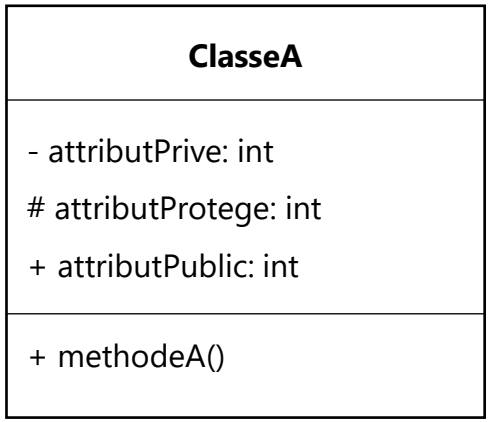
VISIBILITÉ D'UN MEMBRE DEPUIS L'INTÉRIEUR D'UNE CLASSE




```
void ClasseA::methodeA()
{
    attrPrive = 10;      // OK
    attrProtege = 15;    // OK
    attrPublic = 20;     // OK
}
```

ACCÈS SANS RESTRICTION À TOUS LES MEMBRES

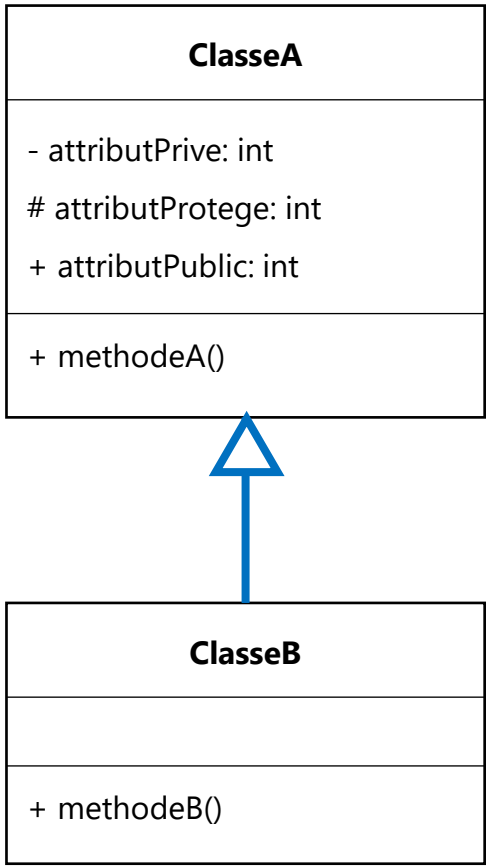
VISIBILITÉ D'UN MEMBRE DEPUIS L'EXTÉRIEUR DE LA CLASSE




```
int main()
{
    ClasseA instance;
    instance.attrPrive = 10;      // ERREUR
    instance.attrProtege = 15;    // ERREUR
    instance.attrPublic = 20;     // OK
}
```

SEULS LES MEMBRES PUBLICS SONT ACCESSIBLES

VISIBILITÉ D'UN MEMBRE DEPUIS UNE CLASSE FILLE



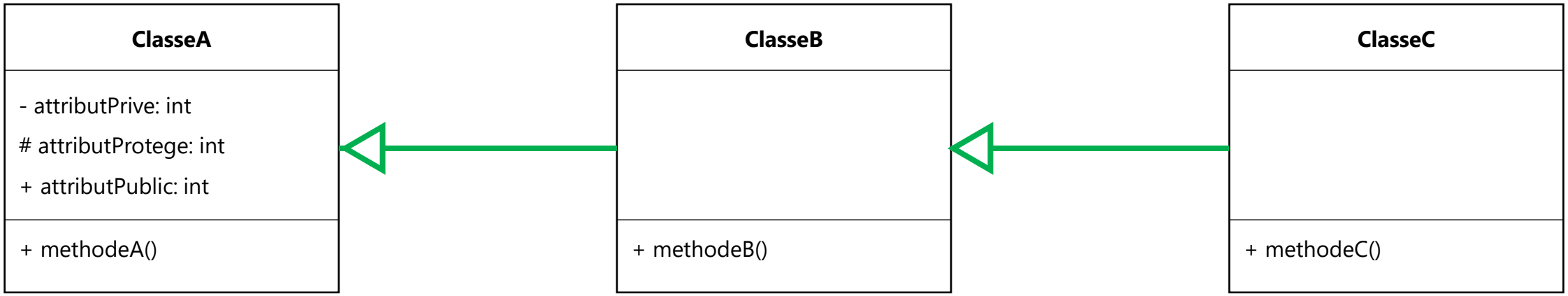

```
void ClasseB::methodeB()
{
    attrPrive = 10;      // ERREUR
    attrProtege = 15;    // OK
    attrPublic = 20;     // OK
}
```

LES MEMBRES PROTÉGÉS ET PUBLICS SONT ACCESSIBLES... ENFIN PRESQUE

MODIFICATEURS D'HÉRITAGE

charles.meunier@u-bourgogne.fr

HÉRITAGE PUBLIC - IMPLÉMENTATION

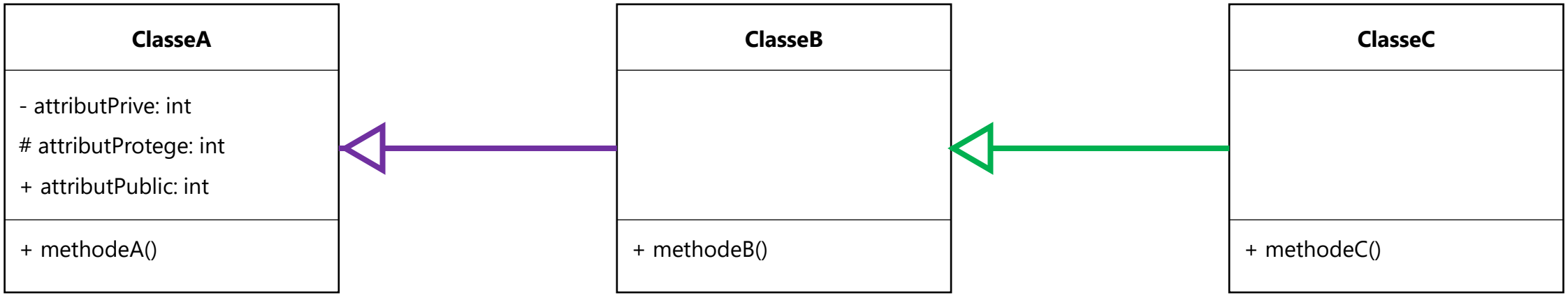



```
class ClasseB : public ClasseA { ... };

class ClasseC : public ClasseB { ... };
```

HÉRITAGE PUBLIC - VISIBILITÉ DES MEMBRES DE CLASSEA

	depuis **ClasseA**	depuis **ClasseB**	depuis **ClasseC**	depuis **main**
- attrPrive	✓	✗	✗	✗
# attrProtege	✓	✓	✓	✗
+ attrPublic	✓	✓	✓	✓

HÉRITAGE PROTÉGÉ - IMPLÉMENTATION

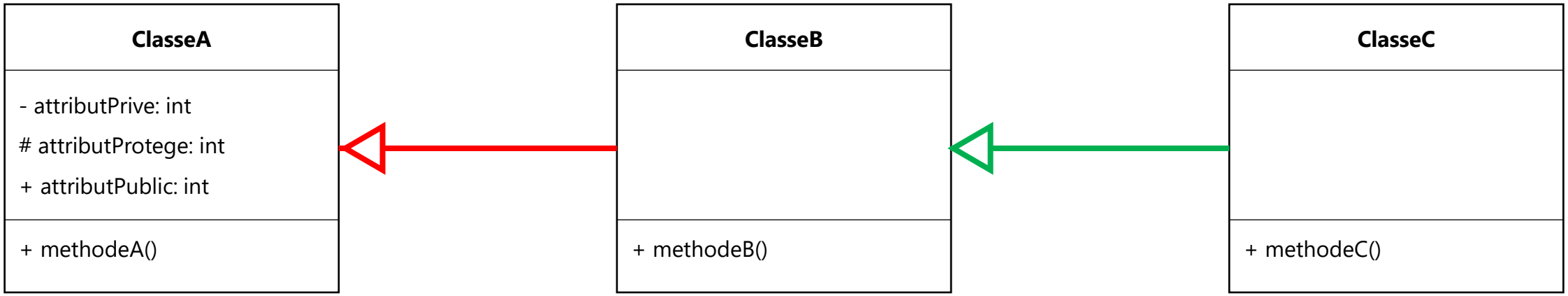



```
class ClasseB : protected ClasseA { ... };

class ClasseC : public ClasseB { ... };
```

HÉRITAGE PROTÉGÉ - VISIBILITÉ DES MEMBRES DE CLASSEA

	depuis **ClasseA**	depuis **ClasseB**	depuis **ClasseC**	depuis **main**
- attrPrive	✓	✗	✗	✗
# attrProtege	✓	✓	✓	✗
+ attrPublic	✓	✓ **=> protégé**	✓	✗

HÉRITAGE PRIVÉ - IMPLÉMENTATION




```
class ClasseB : private ClasseA { ... };

class ClasseC : public ClasseB { ... };
```

HÉRITAGE PRIVÉ - VISIBILITÉ DES MEMBRES DE CLASSEA

	depuis **ClasseA**	depuis **ClasseB**	depuis **ClasseC**	depuis **main**
- attrPrive	✓	✗	✗	✗
# attrProtege	✓	✓ **=> privé**	✗	✗
+ attrPublic	✓	✓ **=> privé**	✗	✗