



LE LANGAGE C++ MASTER 1 LES CLASSES (...*SUITE*)

Jean-Baptiste.Yunes@u-paris.fr
U.F.R. d'Informatique
Université de Paris

2020–2021



LES OBJETS VOLATILES

- Le qualificatif `volatile` permet d'empêcher toute optimisation de compilation (en particulier tout mécanisme de cache de valeur) sur une variable donnée...
 - Une telle variable peut être modifiée à l'insu du programme lui-même
 - ou, modifiée par un autre thread
- Son usage est très restreint et rare...

- un objet peut être qualifié de `volatile` auquel cas
 - les fonctions membres qui peuvent être appelées sûrement sont les méthodes qualifiées de `volatile`

```
class Clock{
private:
    long ticks;
public:
    long getTicksSinceEPOCH() volatile {
        return ticks;
    }
};

int main() {
    volatile Clock *systemClock = reinterpret_cast<Clock *>(0xDEADBEEF);
    cout << systemClock.getTicksSinceEPOCH() << endl;
}
```

LES CONST-VOLATILES

- Ces deux attributs ne sont pas incompatibles!!!!
 - `const` signifie
 - qu'une variable de ce genre est logiquement non-mutable
 - qu'une fonction membre de ce genre peut être appelée sur un objet constant
 - `volatile` signifie
 - qu'une variable de ce genre peut être modifiée à l'insu du programme
 - qu'une fonction membre de ce genre peut être appelée de façon sûre sur un objet volatile

- Ici, on créé un pointeur constant vers un entier volatile constant (un entier modifié par l'environnement mais pas par ce pointeur là).

```
int main() {  
    volatile const int * const mouseButtonState =  
        reinterpret_cast<int *>(0xDEADBABE);  
}
```

MOI, JE...



- Un objet peut parler de lui-même en utilisant le mot-clé `this`
 - permet de lever certaines *ambiguïtés*
 - absolument nécessaire lorsqu'un objet veut agir de son *propre chef* et s'inscrire dans un scénario
- dans un objet constant le type de `this` est

```
const type_de_l_objet *
```

- sinon son type est

```
type_de_l_objet *
```

```
class MaClasse {
    private:
        int valeur;
    public:
        void setValeur(int _valeur) { valeur = _valeur; }
};
```

- levée d'ambiguïté

```
class MaClasse {
    private:
        int valeur;
    public:
        void setValeur(int valeur) { this->valeur = valeur; }
};
```

- moi, je...

```
Liste tousLesObjets;
```

```
class MaClasse {
```

```
public:
```

```
    MaClasse() {
```

```
        tousLesObjets.insert(this);
```

```
    }
```

```
    ~MaClasse() {
```

```
        tousLesObjets.remove(this);
```

```
    }
```

```
};
```

LES MEMBRES STATIQUES

- La classe elle-même peut-être vue comme unité d'encapsulation
 - elle n'est plus vue comme une construction syntaxique mais comme un moule de fabrication
 - elle est une « structure » au sens du stockage
- Ainsi une **classe** peut posséder
 - des données membres (champs)
 - des fonctions membres (méthodes)

- Les membres (données ou fonctions) appartiennent à la classe et non à une quelconque instance particulière
- Les membres sont donc partagés par toutes les instances
- Ces membres sont comme des variables ou fonctions globales mais **encapsulées** dans la classe

- le mot réservé `static` permet de qualifier un membre comme membre de classe
- pour une donnée membre sa déclaration prend la forme :

```
class UneClasse{  
    private:  
        static int unAttribut;  
};
```

- l'initialisation d'une donnée membre statique (ou donnée membre de classe) se fait par une définition au niveau global :

```
int UneClasse::unAttribut = 24;
```

- Pas de surprise : c'est bien une variable globale (encapsulée)

- pour une fonction membre statique sa déclaration prend la forme :

```
class UneClasse{  
    public:  
        static int getAttribut();  
};
```

- et sa définition :

```
int UneClasse::getAttribut() {  
    return unAttribut;  
}
```

- Naturellement une fonction membre statique (ou méthode de classe) ne peut utiliser `this`, puisque `this` n'a de sens que dans le contexte d'un objet (d'une instance de la classe)
- Les membres statiques existent en dehors même de l'existence d'un quelconque instance
- Les membres statiques existent au « démarrage » du programme

- La syntaxe de nommage pour l'utilisation d'un membre statique est :

```
int main() {  
    UneClasse::unAttribut = 23;  
    UneClasse::getAttribut();  
}
```

- Par « extension », il est autorisé d'appeler un membre statique en partant d'une instance :

```
int main() {  
    UneClasse unObjet;  
    unObjet.unAttribut = 23; // C'est partagé  
    unObjet.getAttribut(); // C'est commun  
}
```

Non recommandé

- Note technique :
 - Attention en C++ les classes ne sont pas des objets à part entière...
 - C'est pourquoi on utilise la notation `::` pour accéder aux membres statiques...
- En dehors de `static` les membres peuvent être qualifiés autrement `const`, `volatile`

- À propos de l'initialisation des membres statiques constants

- constants :

```
class Couleur {  
    public:  
        static const int ROUGE = 0xff0000;  
};
```

Forme possible

- OU

```
class Couleur {  
    public:  
        static const int ROUGE;  
};
```

Forme recommandée

```
const int Couleur::ROUGE = 0xff0000;
```

- Usages fréquents des données membres statiques :
 - définition de constantes globales à la classe
 - dénombrement d'instances, liste d'instances
- Usages fréquents des fonctions membres statiques :
 - accesseurs d'attributs statiques
 - fonctions utilitaires

```
class Calcul {  
    public:  
        static const double PI = 3.1415926;  
        static int addition(int a,int b):  
        static int soustraction(int a,int b);  
};
```

```
int main() {  
    Calcul::addition(6,5);  
    Calcul::soustraction(4,3);  
  
    Calcul c;  
}
```



C'est louche ? Non ?

- **Nous réglerons ce problème plus tard...**



LA PROTECTION LE CONTRÔLE D'ACCÈS



- Rappel : l'encapsulation c'est
 - une boîte contenant des éléments
 - la classe comme structure de donnée et fonctions agissant dessus ou grâce à
 - un contrôle d'accès aux éléments
 - un mécanisme permettant de cacher certains secrets de fabrication ou de fonctionnement

- La protection est constituée en domaines
- Un membre appartient dernier domaine déclaré qui apparaît avant sa propre déclaration
- Le domaine par défaut pour une classe est le domaine privé (`private`)

```
class UneClasse {  
    déclarations  
    domaine :  
    déclarations  
    domaine :  
    déclarations  
};
```

- Le domaine privé, mot réservé `private` :
 - les éléments de ce domaine ne sont atteignables que depuis des fonctions membres (statiques ou non) de la classe, *i.e.* le contexte autorisant le nommage ne peut être que celui d'une méthode de la classe (statique ou non)

```
void A::methode1() {  
    this->attribut = 45;  
}  
void A::methode2() {  
    attribut = 666;  
}  
int main() {  
    A a;  
    a.attribut;  
}
```

dans A

dans A

dans main() qui n'est pas dans A!

```
class A {  
    private:  
        int attribut;  
        void methode1();  
    public:  
        void methode2();  
}
```

- Le domaine privé, mot réservé `private` :
 - on peut créer des classes dans des classes. Il s'agit de classes incluses. La protection joue tout autant. On peut ainsi structurer à l'intérieur d'une structure et bénéficier du contrôle d'accès

```
void A::methode() {  
    A::B b;  
}
```

dans A

```
int main() {  
    A::B b;  
}
```

*dans main() qui n'est pas dans A!
Le type B est donc inutilisable en dehors de A*

```
class A {  
    private:  
        class B {};  
    public:  
        void methode();  
};
```

- Le domaine privé, mot réservé `private` :
 - on peut créer des classes non instanciables...
 - utilité ?
 - une classe module...

```
int main() {  
    Calcul::add(56,78);  
    Calcul c;  
}
```

public

```
class Calcul {  
    private:  
        Calcul() {};  
    public:  
        static int add(int,int);  
};
```

construire c, nécessite l'appel au constructeur qui est privé alors qu'on est dans main() qui n'est pas dans A

- Le domaine privé, mot réservé `private` :
 - on peut créer des classes non instanciables...
 - utilité ?
 - une **usine** (*factory*)...

```
A *A::create() {  
    return new A;  
}
```

ok dans A

```
int main() {  
    A *pa = A::create();  
}
```

on passe forcément par la fonction

```
class A {  
    private:  
        A() {};  
    public:  
        static A *create();  
};
```

- Le domaine privé, mot réservé `private` :
 - on peut créer des classes non instanciables...
 - utilité ?
 - une usine (*factory*) ?
 - un **singleton**...

```
A *A::create() {  
    return &leSeulEtLUnique;  
}  
  
int main() {  
    A *pa1 = A::create();  
    A *pa2 = A::create();  
}
```

```
class A {  
    private:  
        A() {};  
        static A leSeulEtLUnique;  
    public:  
        static A *create();  
};  
  
A A::leSeulEtLUnique;
```


- Le domaine privé, mot réservé `private` :
 - on peut créer des classes non instanciables...
 - utilité ?
 - une usine (*factory*) ?
 - l'unicité...

```
Nbre::Nbre(double v) {
    valeur = v;
}
Nbre *Nbre::create(double v)
    if (v==3.1415926)
        return &PI;
    return new Nbre(v);
}
int main() {
    Nbre *n1 = Nbre::create(3.1415926);
    Nbre *n2 = Nbre::create(3.1415926);
    Nbre *n3 = Nbre::create(6.34);
}
```

```
class Nbre {
private:
    double valeur;
    Nbre(double v);
    static Nbre PI;
public:
    static Nbre *create(double);
};
```

```
Nbre Nbre::PI(3.1415926);
```

- Le domaine public, mot réservé `public` :
 - les éléments de ce domaine sont atteignables depuis n'importe quel point du programme

```
void A::methode1() {  
    attribut = 45;  
}  
void A::methode2() {  
    attribut = 666;  
}  
int main() {  
    A a;  
    a.attribut;  
}
```

dans A

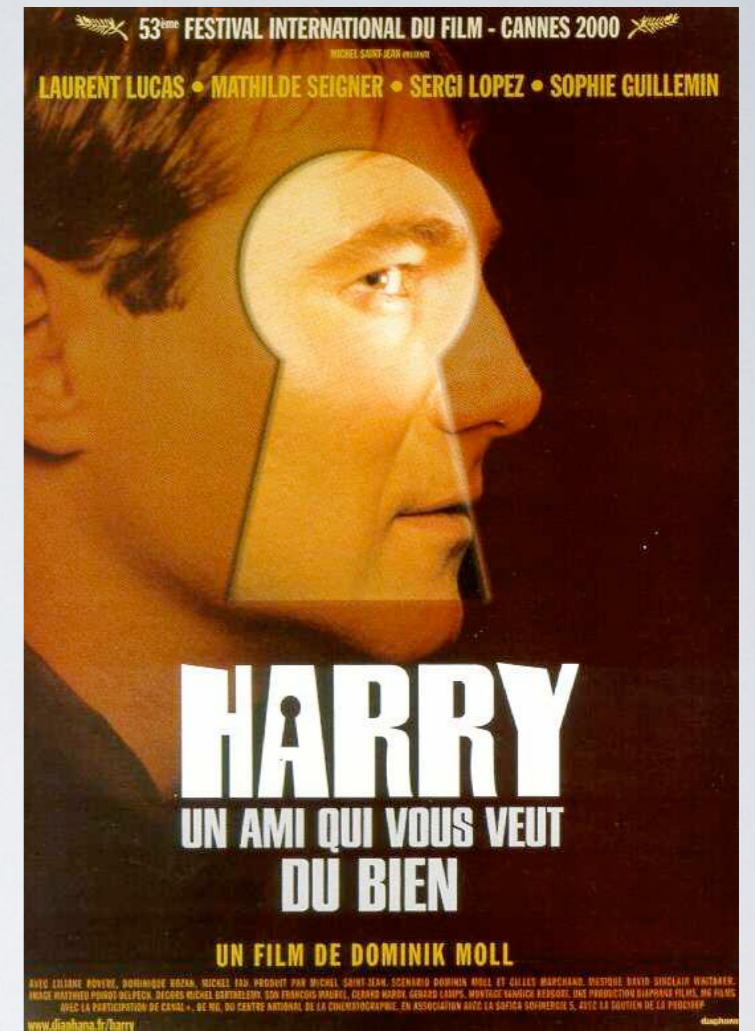
dans A

*dans main() qui n'est pas dans A!
Mais public!*

```
class A {  
    private:  
        void methode1();  
    public:  
        int attribut;  
        void methode2();  
}
```



L'AMITIÉ



- Dans certains cas, les contrôles d'accès s'avèrent trop sévères ou inadéquats :
- On aimerait qu'une fonction (non membre) ou que les instances d'une classe puissent accéder à certains membres privés
- On peut songer à créer des accesseurs publics mais dans ce cas la fonction ne sera pas la **seule** autorisée à accéder aux membres privés!

```
class DistributeurBoisson {
private:
    int gain;
public:
    static const int CAFE      =0;
    static const int THE      =1;
    static const int CHOCOLAT=2;
    static const int SODA     =3;
    Boisson &obtenirUneBoisson(int type,int valeur);
};
```

```
Boisson &DistributeurBoisson::obtenirUneBoisson(int
type,int valeur) {
    ...
    gain += valeur;
    ...
    return ...
}
```

- Comment représenter le *collecteur* ?

```
class DistributeurBoisson {
private:
    int gain;
    int recupererGain();
public:
    Boisson &obtenirUneBoisson(int type,int valeur);
};

class Collecteur {
public:
    void prendsLOseilleEtTireToi(DistributeurBoisson &d);
};

void Collecteur::prendsLOseilleEtTireToi(DistributeurBoisson &d) {
    d.recupererGain();
}
```

Interdit, la méthode est privée

```
class DistributeurBoisson {
    private:
        int gain;
    public:
        int recupererGain();
        Boisson &obtenirUneBoisson(int type, int valeur);
};

class Collecteur {
    public:
        void prendsLOseilleEtTireToi(DistributeurBoisson &d);
};

void Collecteur::prendsLOseilleEtTireToi(DistributeurBoisson &d) {
    d.recupererGain();
}

class Voleur {
    public:
        void prendsLaThuneEtCasseToi(DistributeurBoisson &d) {
            d.recupererGain();
        };
};
```

Ok, super!

Non! Argh!

- La solution est de *lever* la contrainte d'accès pour la fonction ou la classe voulue.
 - La classe qui désire autoriser une entité à accéder à ses membres privés lui en donne la permission
 - Elle lui offre son amitié / apporte sa confiance
- Attention la relation d'amitié n'est pas symétrique
 - Le concept de Distributeur décide que le concept de Collecteur est son ami
 - tout collecteur est l'ami de n'importe quel distributeur
 - pas l'inverse


```
class DistributeurBoisson {
    private:
        int gain;
        int recupererGain();
    public:
        Boisson &obtenirUneBoisson(int type,int valeur);
        friend
            void Collecteur::prendsLOseilleEtTireToi(DistributeurBoisson &);
};
class Collecteur {
    public:
        void prendsLOseilleEtTireToi(DistributeurBoisson &d);
};

void Collecteur::prendsLOseilleEtTireToi(DistributeurBoisson &d) {
    d.recupererGain();
}
```

Ok, Collecteur est un ami...

Problème : le compilateur
n'est pas content

- Le problème est que dans la déclaration du type `DistributeurBoisson` on utilise le type `Collecteur`
 - nécessite donc une déclaration préliminaire du type `Collecteur`
- Dans la déclaration du type `Collecteur` on utilise le type `DistributeurBoisson`
 - nécessite donc une déclaration préliminaire du type `DistributeurBoisson`

```
class DistributeurBoisson;

class Collecteur {
public:
    void prendsLOseilleEtTireToi(DistributeurBoisson d &);
};

class DistributeurBoisson {
private:
    int gain;
    int recupererGain();
public:
    Boisson &obtenirUneBoisson(int type,int valeur);
friend
    void Collecteur::prendsLOseilleEtTireToi(DistributeurBoisson &);
};
```

- Ce mécanisme s'appelle une **déclaration anticipée** (forward declaration)

- Remarques (points à noter) :
 - les déclarations d'amitiés peuvent apparaître dans n'importe quelle section, cela ne joue pas
 - le grain peut être fin ou gros
 - grain fin : amitié pour un ensemble précis de fonctions **Prudence, prévaut : *paranoïd mode***
 - gros grain : amitié pour une classe entière (donc par extension à toutes les fonctions membres de la classe) **Confiance règne**

- grain fin :

```
class A {  
    friend B::trucmuche();  
    friend T::bidule();  
    friend Z::machin();  
};
```

- gros grain :

```
class A {  
    friend class B;  
};
```

INITIALISATION DES MEMBRES

- L'instanciation d'un objet nécessite une création avec initialisation
- Comment faire référence à la bonne initialisation ?

```
class CompteEnBanque {  
    private:  
        int position;  
    public:  
        CompteEnBanque();  
};
```

```
CompteEnBanque::CompteEnBanque() {  
    position = 0;  
}
```

position est affecté
ici pas initialisé!!!

- Situation bien pire...

```
class Point {  
    private:  
        int abscisse, ordonnee;  
    public:  
        Point(int x,int y) { abscisse = x; ordonnee = y; }  
};  
  
class SegmentDeDroite {  
    private:  
        Point premier, second;  
    public:  
        SegmentDeDroite(int x1,int y1,int x2,int y2);  
};
```

l'instanciation du segment
nécessite l'instanciation des
deux points et donc leur
initialisation

- Rappel : l'idée derrière les constructeurs est d'initialiser toutes les variables pour éviter les problèmes...

```
class Point {
private:
    int abscisse, ordonnee;
public:
    Point(int x,int y) { abscisse
};
class SegmentDeDroite {
private:
    Point premier, second;
public:
    SegmentDeDroite(int x1,int y1,int x2,int y2);
};
SegmentDeDroite::SegmentDeDroite(int x1,int y1,int x2,int y2) :
premier(x1,y1), second(x2,y2) {
    ...
}
```

on fait un appel au constructeur adéquat pour chaque donnée membre

```
class Point {
private:
    int abscisse, ordonnee;
public:
    Point(int x=0,int y=0) {
        abscisse = x;
        ordonnee = y;
    }
};
```

```
class SegmentDeDroite {
private:
    Point premier, second;
public:
    SegmentDeDroite(int x2,int y2);
};
```

```
SegmentDeDroite::SegmentDeDroite(int x2,int y2) : second(x2,y2)
{
    ...
}
```

on utilise le constructeur sans paramètre ou avec paramètre mais valeurs par défaut

- La syntaxe s'étend aux données membres de types primitifs (syntaxe fonctionnelle d'initialisation)

```
class Point {
    private:
        int abscisse, ordonnee;
    public:
        Point(int x=0,int y=0) {
            abscisse = x; ordonnee = y;
        }
};

class Cercle {
    private:
        Point centre;
        double rayon;
    public:
        Cercle(int x,int y,double r);
};

Cercle::Cercle(int x,int y,double r) : centre(x,y), rayon(r) {
    ...
}
```

- On notera qu'en C++11 on devra utiliser la syntaxe d'initialisation utilisant les accolades...

```
class Point {
private:
    int abscisse, ordonnee;
public:
    Point(int x=0,int y=0) {
        abscisse = x; ordonnee = y;
    }
};
class Cercle {
private:
    Point centre;
    double rayon;
public:
    Cercle(int x,int y,double r);
};
Cercle::Cercle(int x,int y,double r) : centre{x,y}, rayon{r} {
    ...
}
```

- Ordre des créations et initialisation pour une classe avec des données membres de classe
 - On alloue la mémoire (pour le tout)
 - On appelle les constructeurs adéquats pour toutes les données membres (dans l'ordre de leur apparition dans la déclaration de la classe...)
 - On exécute le code du constructeur de la classe englobante (les données membres sont donc déjà initialisées!)

- Quid des destructions ?
- Les destructions s'opèrent en ordre **exactement** inverse de celui des constructions...

```
class A {
private:
    int valeur;
public:
    A(int x) { valeur = x; cout << valeur << endl; }
    ~A() { cout << '-' << valeur << endl; }
};
```

```
class B {
private:
    A un;
    A deux;
public:
    B(int x,int y) : un(x), deux(y) {};
};
```

```
int main() {
    B b(1,2);
}
```

```
[Trotinette:~] yunes% ./test
1
2
-2
-1
[Trotinette:~] yunes%
```

```
class A {
private:
    int valeur;
public:
    A(int x) { valeur = x; cout << valeur << endl; }
    ~A() { cout << '-' << valeur << endl; }
};
```

```
class B {
private:
    A un;
    A deux;
public:
    B(int x,int y) : deux(x), un(y) {};
};
```

```
int main() {
    B b(1,2);
}
```

```
[Trotinette:~] yunes% ./test
1
2
-2
-1
[Trotinette:~] yunes%
```

on échange mais rien à faire
au mieux, le compilateur avertit du
«problème»


```
class A {
private:
    int valeur;
public:
    A(int x) { valeur = x; cout << valeur << endl; }
    ~A() { cout << '-' << valeur << endl; }
};
```

```
class B {
private:
    A deux;
    A un;
public:
    B(int x,int y) : un(x), deux(y) {};
};
```

```
int main() {
    B b(1,2);
}
```

on échange les déclarations
et tout change!

```
[Trotinette:~] yunes% ./test
2
1
-1
-2
[Trotinette:~] yunes%
```

- Ce problème peut être tracé à la compilation en activant le niveau maximal d'avertissement
-Wall (gcc/llvm)

```
A.cpp:17:20: warning: field 'deux' will be initialized after field 'un'
               [-Wreorder]
    B(int x,int y) : deux(x), un(y) {};
```