ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Classes & C++ : bases

BATTERIES
INCLUDED

COURS 5

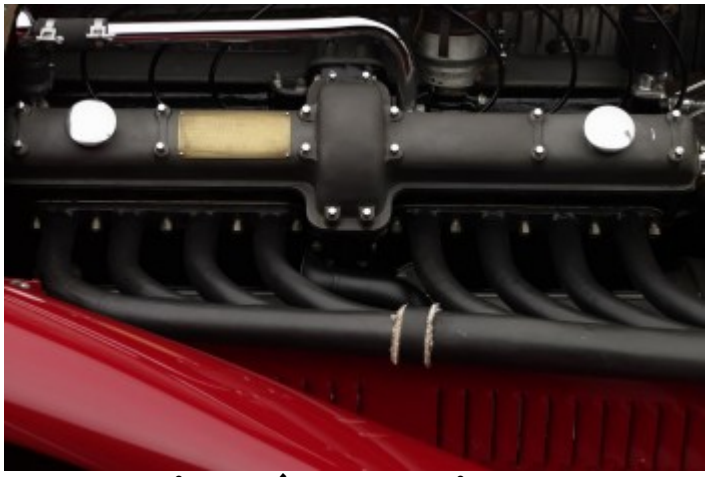
A) La classe en C++
B) L’encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

COURS 5

A) La classe en C++
B) L’encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

La classe en C++

`interface`



`implémentation`

`code client`

La classe en C++

Pour illustrer le format général de la classe en C++ reprenons la ***classe*** *Compte, avec 2* ***attributs***
(modèle très simplifié : plus tard le titulaire sera un objet...)

- *titulaire : chaîne de caractères*
- *solde : une valeur flottante*

et 5 ***méthodes***

- *Créer un compte avec solde initial paramétrable*
- *Libérer un compte*
- *Afficher un compte*
- *Créditer un compte avec crédit en paramètre*
- *Débiter un compte avec débit en paramètre*

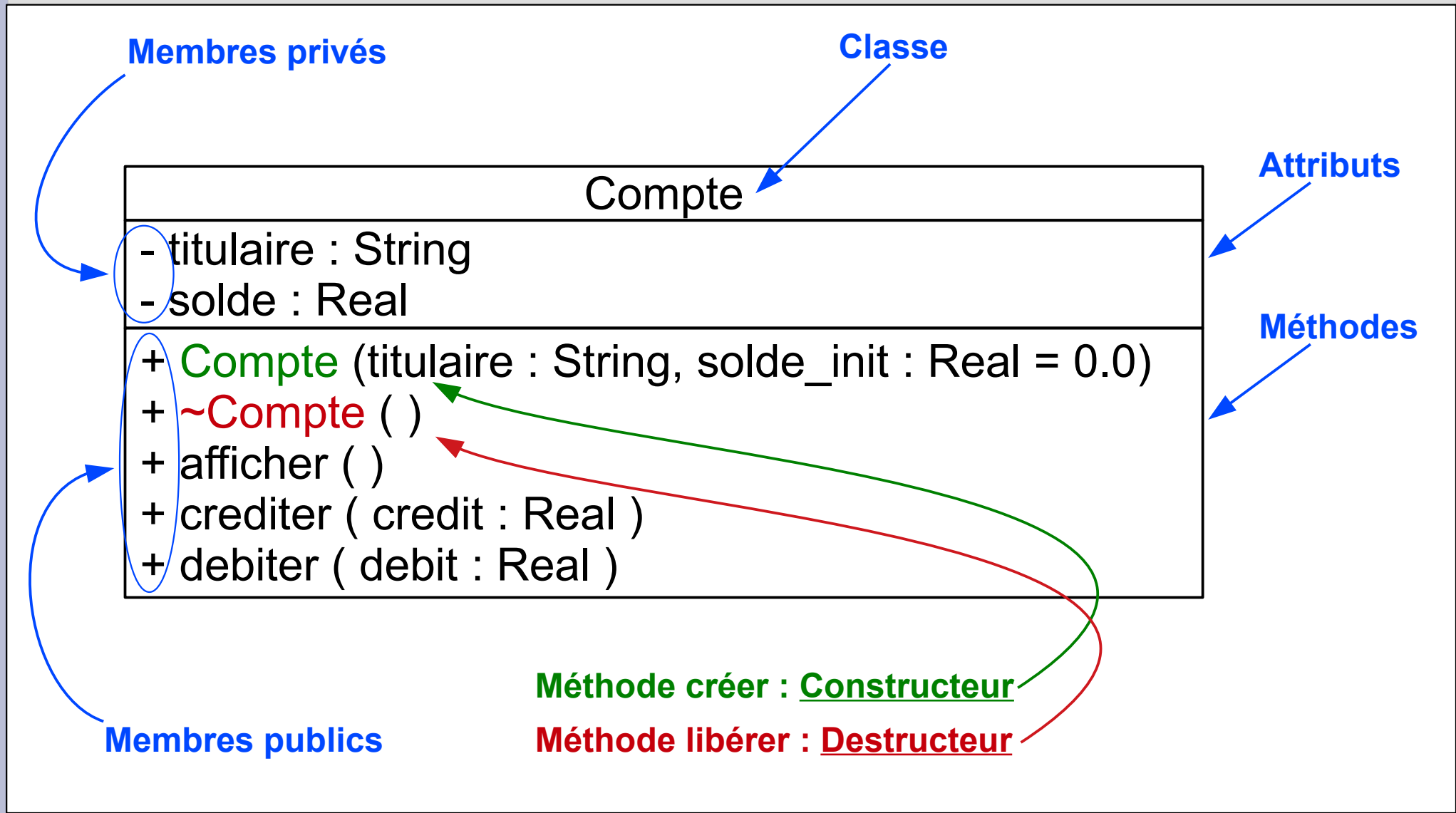
La classe en C++

*La **classe** Compte en notation UML normalisée*

Compte
- titulaire : String - solde : Real
+ Compte (titulaire : String, solde_init : Real = 0.0) + ~Compte ( ) + afficher ( ) + crediter (credit : Real) + debiter (debit : Real)

La classe en C++

*La **classe** Compte en notation UML normalisée*



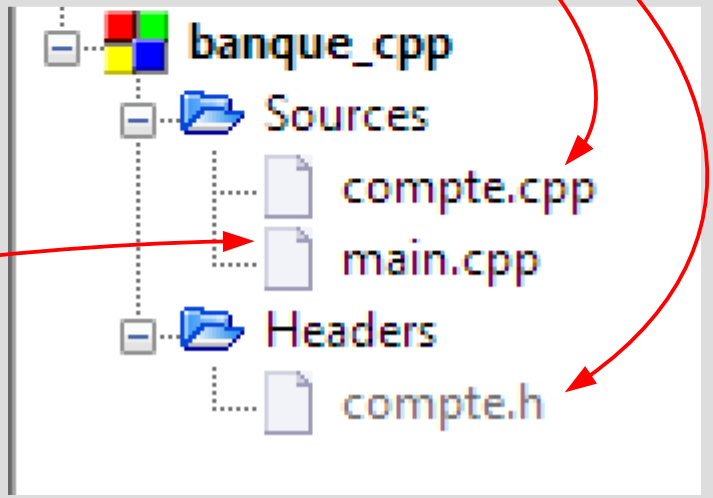
La classe en C++

En C++ on distingue

- *l'interface d'une classe et l'implémentation d'une classe sont le code **utilisé** ou **appelé***

  c'est le code qu'on doit écrire pour développer la classe

- *le code **utilisateur** du type ou code **client** ou appelant*



La classe en C++

- *Attention confusion de terminologie OBJET / C++*
- *En termes de **conception orientée objet***

INTERFACE OBJET
Membres **publiques**
utilisables par le client

Compte
- titulaire : String **- solde : Real**
+ Compte (titulaire : String, solde_init : Real = 0.0) **+ ~Compte ( )** **+ afficher ( )** **+ crediter (credit : Real)** **+ debiter (debit : Real)**

INTERFACE OBJET ≠ INTERFACE C++

La classe en C++

- *Attention confusion de terminologie OBJET / C++*
- *En termes de **C++** interface = déclarations du .h*

compte.h

INTERFACE C++

```
/// Déclaration d'un type "compte en banque"
class Compte
{
    /// Attributs : déclarations
    private :
        std::string m_titulaire;
        float m_solde;


    /// Méthodes : déclarations (prototypes)
    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
        void afficher() const;
        void crediter(float credit);
        void debiter(float debit);

};
```

La classe en C++

- *C'est l'interface OBJET qui doit rester **stable** : la changer conduit à casser le code appelant*

compte.h

```
/// Déclaration d'un type "compte en banque"
class Compte
{
    /// Attributs : déclarations
    private :
        std::string m_titulaire;
        float m_solde;


    /// Méthodes : déclarations (prototypes)
    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
        void afficher() const;
        void crediter(float credit);
        void debiter(float debit);

};
```

Seules les méthodes de l'objet ont accès aux données internes déclarées « private » ceci ne fait pas partie de l'interface OBJET : on peut **changer** des choses ici **sans casser le code client**

Les méthodes publiques constituent l'interface OBJET de la classe : **changer** les formats d'appel de l'interface **casse le code client**

On peut toujours **ajouter** de nouvelle méthode **sans rien casser**

La classe en C++

- *C'est l'interface OBJET qui doit rester **stable** : la changer conduit à casser le code appelant*

compte.h

```
class Compte
{
    private :



    public :


};
```

INTERFACE AU SENS C++
technique : fichier à inclure pour utiliser la classe

INTERFACE AU SENS OBJET
mode d'emploi à respecter pour utiliser la classe

La classe en C++

*Le fichier .cpp donne l'**implémentation** (définitions) des méthodes déclarées dans le fichier.h*

compte.cpp

IMPLÉMENTATION C++

```
#include "compte.h"

#include <iostream>
#include <string>
#include <stdexcept>

/// Méthodes : définitions

Compte::Compte(std::string titulaire, float solde_init)
{ ... }
Compte::~Compte()
{ ... }
void Compte::afficher() const
{ ... }
void Compte::crediter(float credit)
{ ... }
void Compte::debiter(float debit)
{ ... }
std::string Compte::getTitulaire() const
{ ... }
```

La classe en C++

*Le fichier .cpp donne l'**implémentation** (définitions) des méthodes déclarées dans le fichier.h*

compte.cpp

IMPLÉMENTATION
C++

```
void Compte::crediter(float credit)
{ ... }
```

Code d'implémentation,
ceci ne fait pas partie de
l'interface OBJET :
on peut **changer** des choses ici
sans casser le code client
à condition que la méthode
continue de jouer **le même rôle**

COURS 5

A) La classe en C++
B) L’encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

L’encapsulation

L'encapsulation

L'encapsulation est la « protection des données » mais aussi leur dissimulation derrière l'interface

compte.cpp

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

2020

Implémentation d'une 1ère version du logiciel : utilisation d'un attribut pour stocker le solde du client

```
void Compte::crediter(float credit)
{
  /// Create a DATABASE connexion
  sql::Driver* driver = get_driver_instance();
  sql::Connection* con =
       driver->connect("tcp://myBank:3306", "root", "1234");

  ///  Connect to the MySQL ClientAccounts database
  con->setSchema("ClientAccounts");

    ...
    ... m_dbKey
    ...
}
```

2021

Implémentation d'une 2ème version du logiciel : utilisation d'une base de donnée pour stocker le solde du client.

L'attribut m_solde disparaît de la classe, il est remplacé par une clé de table de base de donnée

L'encapsulation

L'encapsulation est la « protection des données » mais aussi leur dissimulation derrière l'interface

compte.h

2020

```
class Compte
{
    private :
        std::string m_titulaire;
        float m_solde;

    public :
       void crediter(float credit);
        ...
};
```

instable

stable

Implémentation d'une 1ère version du logiciel : utilisation d'un attribut pour stocker le solde du client

2021

```
class Compte
{
    private :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
       void crediter(float credit);
        ...
};
```

instable

stable

Implémentation d'une 2ème version du logiciel : utilisation d'une base de donnée pour stocker le solde du client.

L'attribut m_solde disparaît de la classe, il est remplacé par une clé de table de base de donnée

L'encapsulation

Le code client n'est ***pas exposé*** *aux* ***changements de représentation interne des entités***

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->crediter(x);
    ...

    ...
    cpts[recipient]->crediter(y);
    ...

    ...
    cpts[rollback]->crediter(z);
    ...
```

~ 200000 lignes de code

compte.h *2020*

```
class Compte
{
    private :
        std::string m_titulaire;
        float m_solde;

    public :
        void crediter(float credit);
        ...
};
```

compte.h *2021*

```
class Compte
{
    private :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
        void crediter(float credit);
        ...
};
```

L'encapsulation

Le code client n'est ***pas exposé*** *aux* ***changements de représentation interne des entités***

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->crediter(x);
    ...

    ...
    cpts[recipient]->crediter(y);
    ...

    ...
    cpts[rollback]->crediter(z);
    ...
```

~ 200000 lignes de code

compte.h — *2020*

```
class Compte
{
    private :
        std::string m_titulaire;
        float m_solde;

    public :
        void crediter(float credit);
        ...
};
```

compte.h — *2021*

```
class Compte
{
    private :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
        void crediter(float credit);
        ...
};
```

L'encapsulation

Le code client n'est ***pas exposé*** *aux* ***changements de représentation interne des entités***

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->crediter(x);
    ...

    ...
    cpts[recipient]->crediter(y);
    ...

    ...
    cpts[rollback]->crediter(z);
    ...
```

~ 200000 lignes de code

2020 **compte.h**

```
class Compte
{
    private :
        std::string m_titulaire;
        float m_solde;

    public :
        void crediter(float credit);
        ...
};
```

2021 **compte.h**

```
class Compte
{
    private :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
        void crediter(float credit);
        ...
};
```

L'encapsulation

Supposons un non-respect de l'encapsulation des données membres ...

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->m_solde += x;
    ...

    ...
    cpts[recipient]->m_solde += y;
    ...

    ...
    cpts[rollback]->m_solde += z;
    ...
```

~ 200000 lignes de code

2020 **compte.h**

```
class Compte
{
    public :
        std::string m_titulaire;
        float m_solde;

    public :
        // Pas besoin de méthode !
        ...
};
```

2021 **compte.h**

```
class Compte
{
    public :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
        // Pas besoin de méthode ?
        ...
};
```

L'encapsulation

Supposons un non-respect de l'encapsulation des données membres ...

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->m_solde += x;
    ...

    ...
    cpts[recipient]->m_solde += y;
    ...

    ...
    cpts[rollback]->m_solde += z;
    ...
```

~ 200000 lignes de code

compte.h *2020*

```
class Compte
{
    public :
        std::string m_titulaire;
        float m_solde;

    public :
        // Pas besoin de méthode !
        ...
};
```

compte.h *2021*

```
class Compte
{
    public :
        std::string m_titulaire;
        std::uint64_t m_dbKey;

    public :
        // Pas besoin de méthode ?
        ...
};
```

L'encapsulation

Supposons un non-respect de l'encapsulation des données membres ...

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->m_solde += x;
    ...

    ...
    cpts[recipient]->m_solde += y;
    ...

    ...
    cpts[rollback]->m_solde += z;
    ...
```

~ 200000 lignes de code

compte.h — *2020*

```
class Compte
{
    public :
        std::string m_titulaire;
        float m_solde;

    public :
        // Pas besoin de méthode !
        ...
};
```

compte.h — *2021*

```
class Compte
{
    public :
        std::string m_titulaire;
?!      std::uint64_t m_dbKey;

    public :
        // Pas besoin de méthode ?
        ...
};
```

L'encapsulation

Le risque n'est pas que ça passe inaperçu !
Le risque est de passer beaucoup plus de temps...

client.cpp

```
... client(...)
{
    std::vector<Compte*> cpts;

    ...
    cpts[i]->m_solde += x;
    ...

    ...
    cpts[recipient]->m_solde += y;
    ...

    ...
    cpts[rollback]->m_solde += z;
    ...
```

~ 200000 lignes de code
~ 477 occurrences de m_solde
en = en += en -= en == etc...

compte.h — *2020*

```
class Compte
{
    public :
        std::string m_titulaire;
        float m_solde;

    public :
        // Pas besoin de méthode !
        ...
};
```

compte.h — *2021*

```
class Compte
{
    public :
        std::string m_titulaire;
?!      std::uint64_t m_dbKey;

    public :
        // Pas besoin de méthode ?
        ...
};
```

L’encapsulation

- *Résultat du non respect du principe d’encapsulation des données membres*
  - *Ou on va renoncer à faire un changement nécessaire de la représentation interne*
  - *Ou on va perdre beaucoup de temps avec un risque considérable d’introduire des erreurs, courir après les incohérences...*
  - *Dans les 2 cas on perd sur la concurrence*
- *L’encapsulation n’est pas une commodité*
- *C’est la viabilité à moyen et long terme d’un système logiciel complexe en évolution*

L’encapsulation

- *Dans tous les cas on ne doit proposer au client (càd mettre en public) que les **membres** qui ont vocation à rester **stables** à long terme (+20 ans)*
- *L’**expérience** montre que c’est possible pour des méthodes bien conçues, pas pour les attributs*

membre →

accès ↓

	`méthode`	`attribut`
`public`	**OUI** C’est l’interface OBJET	**NON** Mauvaise pratique
`private`	**POSSIBLE** Traitements auxiliaires	**OUI** Données "encapsulées"

L’encapsulation

- *Selon le contexte, l’entreprise, l’expérience, la case attribut publique est plus ou moins **taboue***
- *En C++ une struct **est** une classe en accès publique par défaut : pourquoi garder la struct ?*

membre →

accès ↓

	`méthode`	`attribut`
`public`	**OUI** C’est l’interface OBJET	**struct !** Mauvaise pratique ?
`private`	**POSSIBLE** Traitements auxiliaires	**OUI** Données "encapsulées"

L’encapsulation

- *compatibilité avec C => struct en C++*
- *Elle peut être utilisée après mûre réflexion : pour grouper des données dont on pense qu’elle seront **stables** sur le long terme...*
- *Pour des petits objets « techniques »*

accès ↓

	`méthode`	`attribut`
`public`	**OUI** C’est l’interface OBJET	**peut-être (struct)** Avec circonspection
`private`	**POSSIBLE** Traitements auxiliaires	**OUI** Données "encapsulées"

L’encapsulation

- *Mais attention il n’y a pas qu’un problème de stabilité des attributs: il y a aussi un problème de **cohérence** des données*
- *L’usage systématique d’une interface composée de méthode publique peut la garantir...*
- *Pour trouver la date du lendemain vous préférez bidouiller directement les attributs jour/mois/année d’un objet struct Date ? Les mois à 30 à 31 jours ? Le mois de février ? Les année bissextiles ? L’internationalisation ?*
- *Ou passer par la méthode nextDay de class Date*

```
Date demain = aujourdhui.nextDay( );
```

L'encapsulation

*Il est possible (fréquent) d'avoir des méthodes privées : traitements auxiliaires **internes** ...*

compte.h

```
class Compte
{
    /// Attributs
    private :
        std::string m_titulaire;
        std::uint64_t m_dbKey;
        sql::Connection* m_sqlConnex;

    /// Méthodes
    private :
        void ouvrirConnexionDB(); // Utile pour crediter et debiter

    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
        void afficher() const;
        void crediter(float credit);
        void debiter(float debit);
};
```

En 2021 cette méthode est utile aux autres méthodes de la classe. Peut-être qu'en 2024 elle disparaîtra. Et elle ne correspond pas à un service direct qu'un objet Compte joue logiquement pour le code client. Elle ne fait donc pas partie de l'interface OBJET, elle est privée !

L'encapsulation

En dehors des cas particuliers :

Les fonctions membres (méthodes) sont publiques
→ en tout cas les méthodes qui intéressent le client

Les données membres (attributs) sont privées

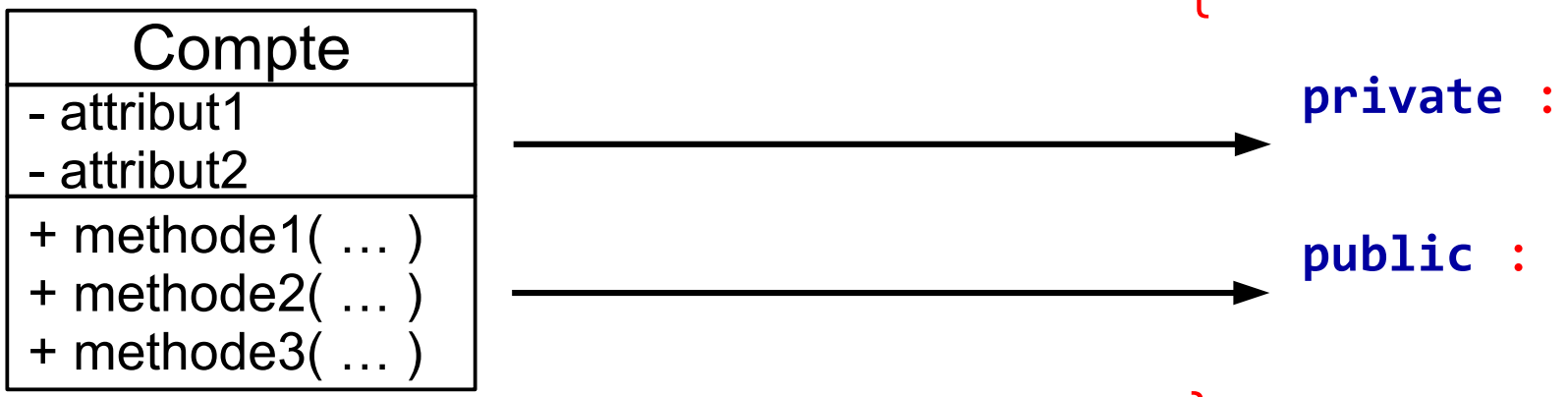
	`méthode`	`attribut`
`public`	**OUI** **C'est l'interface OBJET**	**peut-être (struct)** Avec circonspection
`private`	**POSSIBLE** Traitements auxiliaires	**OUI** **Données "encapsulées"**

L'encapsulation

L'ordre de déclaration des sections private/public n'a pas d'importance techniquement...

Notation « naturelle »
on utilise le même ordre
que la notation UML

compte.h



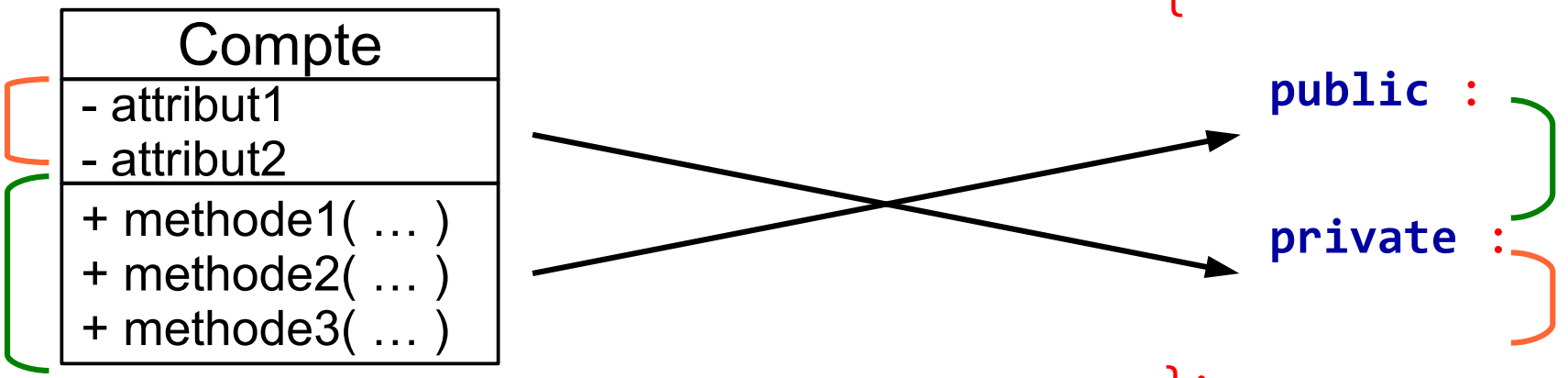
Pour faciliter le passage du modèle UML au C++
pour l'instant c'est l'ordre de déclaration qui sera utilisé

L'encapsulation

L'ordre de déclaration des sections private/public n'a pas d'importance techniquement mais...

Notation recommandée à terme l'interface OBJET en 1er !

compte.h



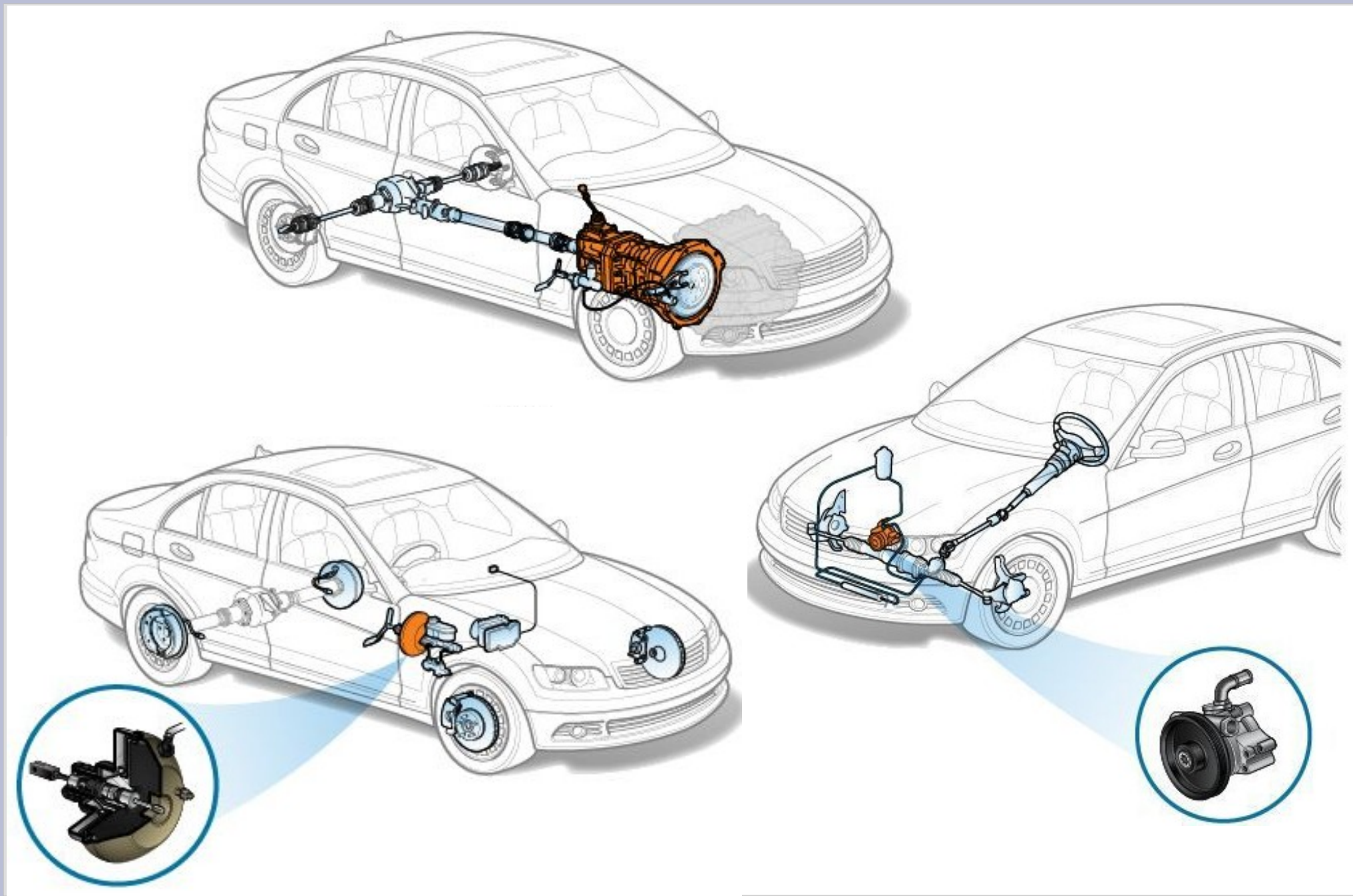
*Quand on fait le modèle objet il est fréquent de commencer d'abord par **les données**... La classe UML présente les données au dessus des **méthodes***

*Quand on fait #include pour utiliser la classe on s'intéresse avant tout au « mode d'emploi » donc au met l'**interface OBJET** en 1er et on "enterre" les **attributs privés***

COURS 5

A) La classe en C++
B) L'encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Les méthodes, this, const



Les méthodes, this, const

Les méthodes sont comme des sous-programmes avec un paramètre implicite : l'objet cible de l'appel

compte.h

```
/// Déclaration d'un type "compte en banque"
class Compte
{
    /// Attributs
    private :
        std::string m_titulaire;
        float m_solde;


    /// Méthodes
    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
        void afficher() const;
        void crediter(float credit);
        void debiter(float debit);

};
```

Les méthodes, this, const

Les méthodes sont <u>comme des sous-programmes</u> avec un paramètre implicite : l'objet cible de l'appel

compte.h

```
/// Déclaration d'un type "compte en banque"
class Compte
{
    /// Attributs
    private :
        std::string m_titulaire;
        float m_solde;


    /// Méthodes
    public :
Constructeur→ Compte(std::string titulaire, float solde_init=0);
Destructeur→  ~Compte();
        void afficher() const;
        void crediter(float credit);
        void debiter(float debit);

};
```

Données membres préfixées par m_
convention utile pour éviter les confusions

Valeur par défaut d'un paramètre

Paramètres, préfixés par _
ou non préfixés (plus léger)

L'objet de type Compte, cible de l'appel,
n'est pas mentionné explicitement :
il est transmis **implicitement** à la méthode

Ici pas de préfixe
Compte::

Les méthodes, this, const

L'implémentation d'une méthode dans le cpp

compte.cpp

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

Compte::
Opérateur de résolution de portée

Les attributs privés inaccessible pour le code client de la classe sont accessible aux méthodes de la classe

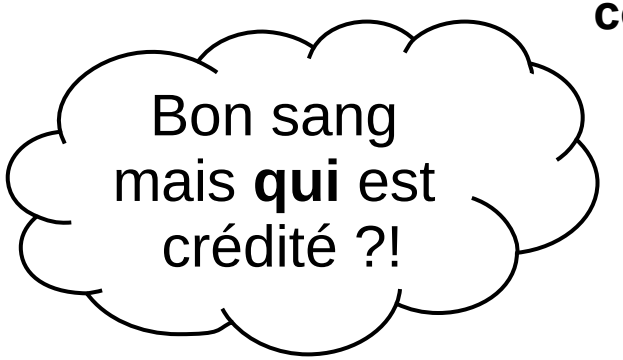
L'objet de type Compte, cible de l'appel, n'est pas mentionné explicitement : il est transmis **implicitement** à la méthode

Les méthodes, this, const

*L'implémentation d'une méthode dans le cpp : on traite **un objet Compte à la fois** ! Lequel ?*

compte.cpp

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```



Les méthodes, this, const

*On traite **un objet Compte à la fois :***
celui qui sert de cible à l'appel de méthode

Code appelé **compte.cpp**

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

Code appelant **client.cpp**

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(20);
    b.crediter(30);
```

Les méthodes, this, const

*On traite **un objet Compte à la fois :***
celui qui sert de cible à l'appel de méthode

Code appelé **compte.cpp**

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

this object

Code appelant **client.cpp**

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(20);
```

Les méthodes, this, const

*On traite **un objet Compte à la fois :***
celui qui sert de cible à l'appel de méthode

Code appelé **compte.cpp**

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

this object

Code appelant **client.cpp**

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};


   b.crediter(30);
```

Les méthodes, this, const

*On traite **un objet Compte à la fois :***
celui qui sert de cible à l'appel de méthode

Code appelé **compte.cpp**

```
void Compte::crediter(float credit)
{
    m_solde += credit;
}
```

1er appel, on modifie le solde du compte **a**

2ème appel, on modifie le solde du compte **b**

Code appelant **client.cpp**

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(20);
    b.crediter(30);
```

Les méthodes, this, const

*Ça fonctionne **comme si** l'appelant envoyait **par adresse une struct à modifier***

```C++
void Compte::crediter
    ( float credit)
{
    m_solde += credit;
}
```

C++

```C
void crediter(Compte* this,
     float credit)
{
    this->m_solde += credit;
}
```

C

```C++
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(20);
    b.crediter(30);
```

C++

```C
void client()
{
    Compte a={"Zig", 0};
    Compte b={"Zag", 0};

    crediter(&a, 20);
    crediter(&b, 30);
```

C

Les méthodes, this, const

*L'**objet de cible** de l'appel est donc bien passé à l'appelant : c'est le « paramètre » **this** implicite*

```
void Compte::crediter
    (float credit)
{
   m_solde += credit;
}
```
Objet cible implicite

Méthode C++ : this objet cible du traitement est implicite **C++**

```
void crediter(Compte* this,
     float credit)
{
    this->m_solde += credit;
}
```
Désignation explicite objet cible

Fonction C : this objet cible du traitement est explicite **C**

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(20);
    b.crediter(30);
}
```
Cibler un objet **C++**

```
void client()
{
    Compte a={"Zig", 0};
    Compte b={"Zag", 0};

    crediter(&a, 20);
    crediter(&b, 30);
}
```
Passage par adresse explicite **C**

Les méthodes, this, const

*Si ce « paramètre » **this** est implicite comment le qualifier, par exemple le rendre **const** ?*

```cpp
void Compte::afficher() const
{
    std::cout << m_solde;
    ...
}
```

C++

```c
void afficher(const Compte* this)
{
    printf("%f", this->m_solde);
    ...
}
```

C

```cpp
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.afficher();
    b.afficher();
```

C++

```c
void client()
{
    Compte a={"Zig", 0};
    Compte b={"Zag", 0};

    afficher(&a);
    afficher(&b);
```

C

Les méthodes, this, const

En qualifiant la méthode de **`const`** *: la qualification est indiquée* ***après*** *le prototype (.h et .cpp)*

```
void Compte::afficher() const
{
    std::cout << m_solde;
    ...
    ...
}
```

Pas de param. à qualifier !

L'objet cible est constant !

C++

```
void afficher(const Compte* this)
{
    printf("%f", this->m_solde);
    ...
    ...
}
```

L'objet cible est constant !

C

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.afficher();
    b.afficher();
```

C++

```
void client()
{
    Compte a={"Zig", 0};
    Compte b={"Zag", 0};

    afficher(&a);
    afficher(&b);
```

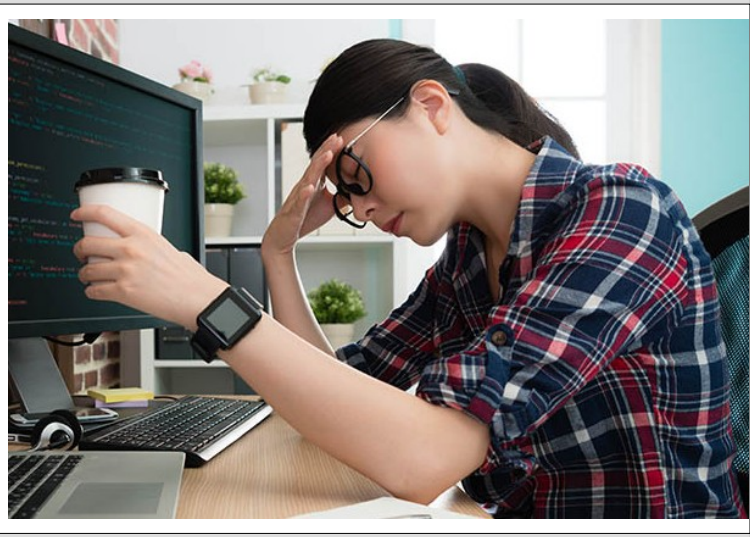
C

Les méthodes, this, const

En qualifiant la méthode de **`const`** *on interdit toute modification des données de l'objet cible*

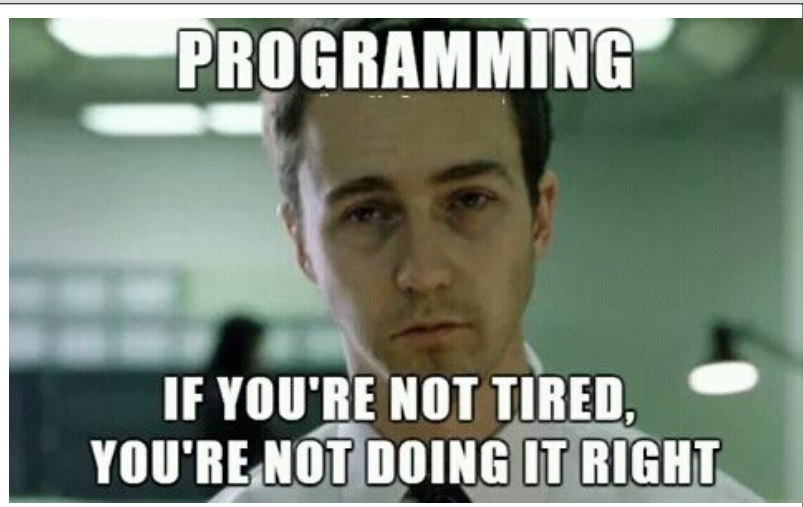
```
void Compte::afficher() const
{
    std::cout << m_solde;
    if ( m_solde=0 ) std::cout << "Vide!";
    ...
}
```

error: assignment of member 'Compte::m_solde' in read-only object



```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.afficher();
    b.afficher();
```



Les méthodes, this, const

Le pointeur **`this`** *sur l'objet cible de l'appel n'est pas une fiction : il est utilisable explicitement*

compte.cpp

```
void Compte::afficher() const
{
    std::cout << this->m_solde;
    ...
    std::cout << this; // Debug : affichage adresse objet cible
}
```

Utilisation explicite, pas utile ici mais possible

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.afficher();
    b.afficher();
```

A chaque appelle « le compilateur » renseigne automatiquement le pointeur `this` *de la méthode avec l'adresse de l'objet cible*
`this` *est de type Classe* (Compte* sur cet exemple)*

Les méthodes, this, const

Fonctions : l'objet peut être passé par valeur (copie) par référence (pas de copie) par adresse (pas de copie)

intermediaire.cpp

```
// Par valeur
void testVal(Compte c)      {    c.afficher();    }

// Par référence
void testRef(Compte& refC) {    refC.afficher();    }

// Par adresse
void testPtr(Compte* ptrC) {    ptrC->afficher();    }
```

client.cpp

```
void client()
{
    Compte a{"Zig", 0};

    testVal(a);

    testRef(a);

    testPtr(&a);
}
```

Attention aux notations spécifiques pointeurs
Méthode pointée ->
Adresse de l'objet &

Les méthodes, this, const

Passage d'objet en paramètre de méthode : pas de règles particulière, idem que fonctions

compte.cpp

```
// Créditer un objet depuis un autre compte
void Compte::crediter(float credit, Compte& debiteur)
{
    if ( debiteur.m_solde>=credit )
    {
        debiteur.m_solde -= credit;
        m_solde += credit;
    }/// else what ?
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(30);     // Zig a 30
    b.crediter(20, a); // Zag prend 20 a Zig : Zag a 20, Zig a 10
```

Les méthodes, this, const

Comme pour les fonctions, les méthodes peuvent être surchargées (on ne se gênera pas)

compte.cpp

```
// Créditer un objet depuis un autre compte
void Compte::crediter(float credit, Compte& debiteur)
{
    if ( debiteur.m_solde>=credit )
    {
        debiteur.m_solde -= credit;
        m_solde += credit;
    }/// else what ?
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(30);     // Zig a 30
    b.crediter(20, a); // Zag prend 20 a Zig : Zag a 20, Zig a 10
```

Les méthodes, this, const

En C++ (idem Java, C#...) les restrictions d'accès sont de niveau classe, pas de niveau objet (smalltalk)

compte.cpp

```
// Créditer un objet depuis un autre compte
void Compte::crediter(float credit, Compte& debiteur)
{
    if ( debiteur.m_solde>=credit )
    {
        debiteur.m_solde -= credit;
         m_solde += credit;
    }/// else what ?
}
```

*Zag peut non seulement lire mais aussi modifier directement les **attributs privés** de Zig*

Les différentes instances d'une même classe sont intimes entre elles
Chouette ! Est-ce une bonne idée ?

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(30);     // Zig a 30
    b.crediter(20, a); // Zag prend 20 a Zig : Zag a 20, Zig a 10
```

Les méthodes, this, const

Même si une méthode a accès direct aux datas de **this** *et des objets de la même classe, on réfléchit !*

compte.cpp

```
void Compte::crediter(float credit, Compte& debiteur)
{
    if ( debiteur.solvable(credit) )
    {
        debiteur.debiter(credit);
        crediter(credit);
    }
    /// else what ?
}
```

Moins de promiscuité avec les attributs privés de this et de l'objet reçu en paramètre :
L'ambiance est plus saine

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(30);     // Zig a 30
    b.crediter(20, a); // Zag prend 20 a Zig : Zag a 20, Zig a 10
```

Les méthodes, this, const

Eviter de manipuler directement les attributs privés conduit à décrire de nombreux « comportements »

compte.cpp

```
// Le compte peut-il débiter une certaine somme ?
bool Compte::solvable(float montant)
{
    return montant <= m_solde;
}
```

En programmation objet on n'écrit pas une nouvelle méthode parce qu'il y a beaucoup de choses à y faire. De nombreuses méthodes seront très courtes mais offrent un meilleur profil d'utilisation pour le code client...
Et la classe elle même est sa 1ère cliente !

compte.cpp

```
void Compte::crediter(float credit, Compte& debiteur)
{
    if ( debiteur.solvable(credit) )
    {
        debiteur.debiter(credit);
        crediter(credit);
    }
    /// else what ?
}
```

Les méthodes, this, const

Finalement quand on veut manipuler "symétriquement" 2 objets on n'utilisera pas une méthode mais une fonction

compte.cpp

```
// Opération de débiter depuis un compte debiteur
// en créditant vers un autre compte beneficiaire
/// Pas une méthode mais presque !
/// Fonction fortement associée à la classe Compte :
/// prototyper dans compte.h, implémenter dans compte.cpp
void transferer(Compte& debiteur,
                Compte& beneficiaire,
                float montant)
{
    /// Plus clair !
    if ( debiteur.solvable(montant) )
    {
        debiteur.debiter(montant);
        beneficiaire.crediter(montant);
    }
    /// else what ?
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 0};
    Compte b{"Zag", 0};

    a.crediter(30);

    // 20€ de Zig à Zag !
    transferer(a, b, 20);
```

Les méthodes, this, const

Quick-and-dirty *dev. of a class : all in main.cpp*

Dès que ça fait plus de 50 lignes on sépare

main.cpp

```
class Thing
{
    private :
        std::string       m_stuff;
        std::vector<int>  m_moreStuff;
        ...

    public :
        char doThat (OtherThing& z);
        void doThere(int x, int y);
        ...
};

char Thing::doThat (OtherThing& z)
{
    ...
}

void Thing::doThere(int x, int y)
{
    ...
}

int main()
{
    Thing myThing{...};
    myThing.doThere(6, 18);
```

possible

≠

recommandé

compte.h

```
class Thing
{
    private :
        std::string       m_stuff;
        std::vector<int>  m_moreStuff;
        ...

    public :
        char doThat (OtherThing& z);
        void doThere(int x, int y);
        ...
};
```

compte.cpp

```
char Thing::doThat (OtherThing& z)
{
    ...
}

void Thing::doThere(int x, int y)
{
    ...
}
```

main.cpp

```
int main()
{
    Thing myThing{...};
    myThing.doThere(6, 18);
```

Les méthodes, this, const

Séparer interface / implémentation / code client

compte.h

```
#include "OtherThing.h"
#include <string>
#include <vector>

class Thing
{
    private :
        std::string       m_stuff;
        std::vector<int>  m_moreStuff;
        ...

    public :
        char doThat (OtherThing& z);
        void doThere(int x, int y);
        ...
};
```

interface

compte.cpp

```
#include "compte.h"
#include <iostream>
...

char Thing::doThat (OtherThing& z)
{
    ...
}

void Thing::doThere(int x, int y)
{
    ...
}
```

implémentation

client.cpp

```
#include "compte.h"
#include <iostream>
...

void client()
{
    Thing myThing{...};
    myThing.doThere(6, 18);
    ...
```

code client

Les méthodes, this, const

Format général appelé / appelant

Code appelé **classe.cpp**

```
TypeRetour Classe::methode(Type1 param1, … ) const ou pas
{
    …  m_attribut1 …  param1 …
    …  m_attribut2 …  if else for while …
    …  return …
}
```

Code appelant **client.cpp**

```
objetCible.methode(param1, … )
```

COURS 5

A) La classe en C++
B) L'encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Cycles de vie des objets

Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **automatiques à privilégier***

```
void Client()
{
    Compte a{...};

    Compte b{...};


    if (...)
    {
        Compte e{...};


    }


}
```

La destruction des objets automatiques est automatique !
*A la fin du **scope** l'objet est détruit*
Fin scope = fermeture bloc { } ou return

*L'**ordre** de construction est toujours celui des déclarations*

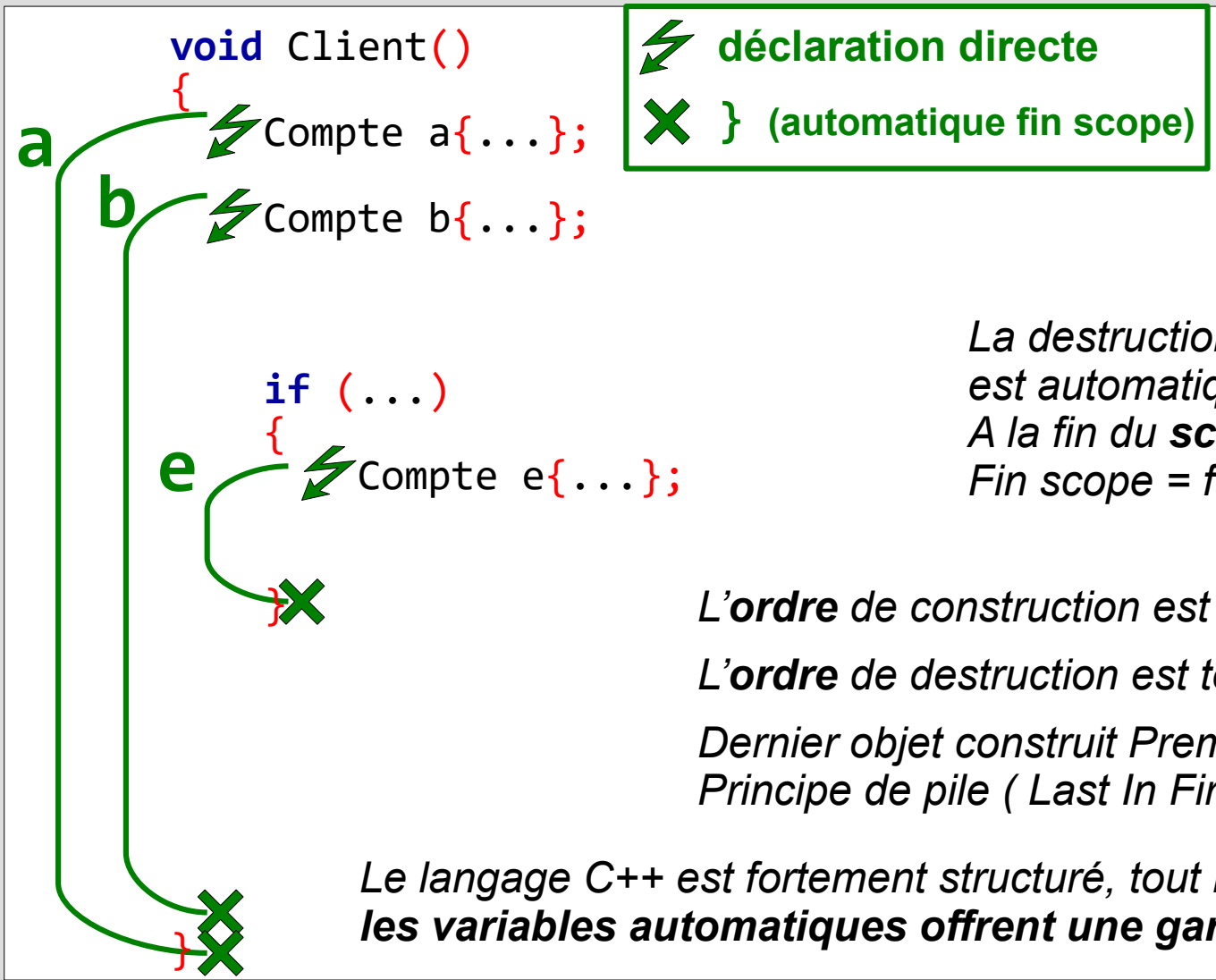
*L'**ordre** de destruction est toujours l'ordre inverse*

Dernier objet construit Premier détruit !
Principe de pile (Last In First Out)

Le langage C++ est fortement structuré, tout bloc ouvert doit être fermé :
les variables automatiques offrent une garantie forte de ne pas fuir ...

Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **automatiques** **structurellement safe !***



La destruction des objets automatiques est automatique !
*A la fin du **scope** l'objet est détruit*
Fin scope = fermeture bloc { } ou return

*L'**ordre** de construction est toujours celui des déclarations*

*L'**ordre** de destruction est toujours l'ordre inverse*

Dernier objet construit Premier détruit !
Principe de pile (Last In First Out)

Le langage C++ est fortement structuré, tout bloc ouvert doit être fermé :
les variables automatiques offrent une garantie forte de ne pas fuir ...

Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **dynamiques peuvent fuir facilement***

*Objets dynamiques sont **persistants**, ils survivent à la fin du scope*
*La construction d'un objet dynamique est manuelle avec **new***
*La destruction d'un objet dynamique est manuelle avec **delete***
*Oubli de **delete** et perte du pointeur => **fuite mémoire***

```
void Client()
{
    Compte* c = new Compte{...};
    Compte* d = nullptr;

    if (...)
    {
        d = allouer(...);
    }

    delete c;

    if (...)
        liberer(d);

}
```

```
Compte *allouer(...)
{
    Compte* r;
    ...
    r = new Compte{...};
    ...
    return r;
}
```

```
void liberer(Compte *s)
{
    ...
    if ( s!=nullptr )
        delete s;
    ...
}
```

Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **dynamiques peuvent fuir facilement***

```
void Client()
{
    Compte* c = new Compte{...};
    Compte* d = nullptr;

    if (...)
    {
        d = allouer(...);
    }

    delete c;

    if (...)
        liberer(d);

}
```

Ici l'utilisation de l'allocation dynamique présente peu d'intérêt objet automatique serait préférable ici !

*c

Si on oublie le delete l'espace mémoire d'un objet Compte est perdu : sizeof(Compte) octets

new

delete (manuellement)

```
Compte *allouer(...)
{
    Compte* r;
    ...
    r = new Compte{...};
    ...
    return r;
}


void liberer(Compte *s)
{
    ...
    if ( s!=nullptr )
        delete s;
    ...
}
```

Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **dynamiques : entités persistantes***

*Ici l'objet a un **cycle de vie complexe***
Il doit survivre aux bloc fermants
Il est peut-être créé, peut-être pas
On veut éviter de copier ses données

new
delete (manuellement)

```
void Client()
{
    Compte* c = new Compte{...};
    Compte* d = nullptr;

    if (...)
    {
        d = allouer(...);
    }

    delete c;

    if (...)
        liberer(d);

}
```

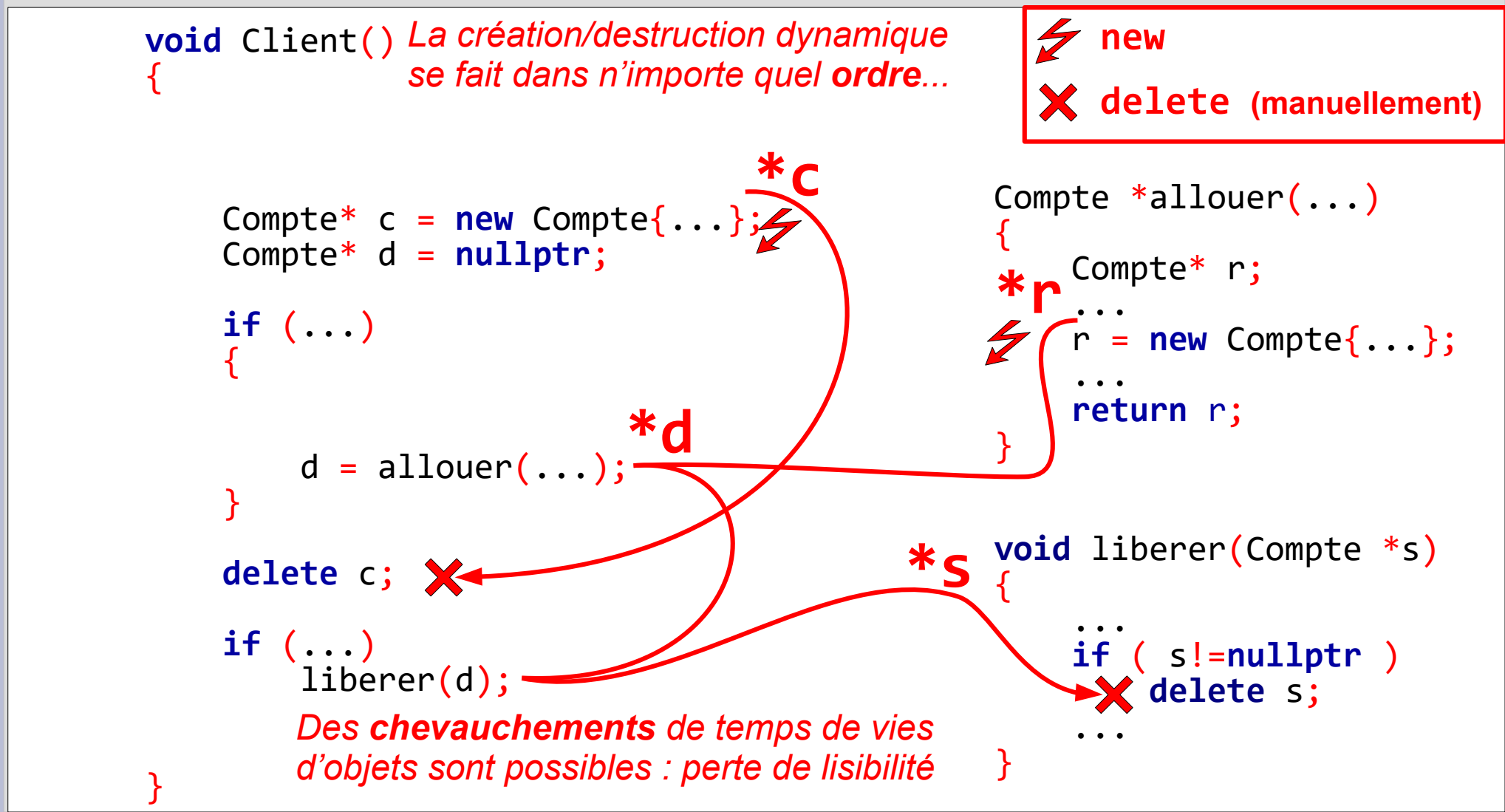
```
Compte *allouer(...)
{
    Compte* r;
    ...
    r = new Compte{...};
    ...
    return r;
}
```

```
void liberer(Compte *s)
{
    ...
    if ( s!=nullptr )
        delete s;
    ...
}
```

*r *d *s

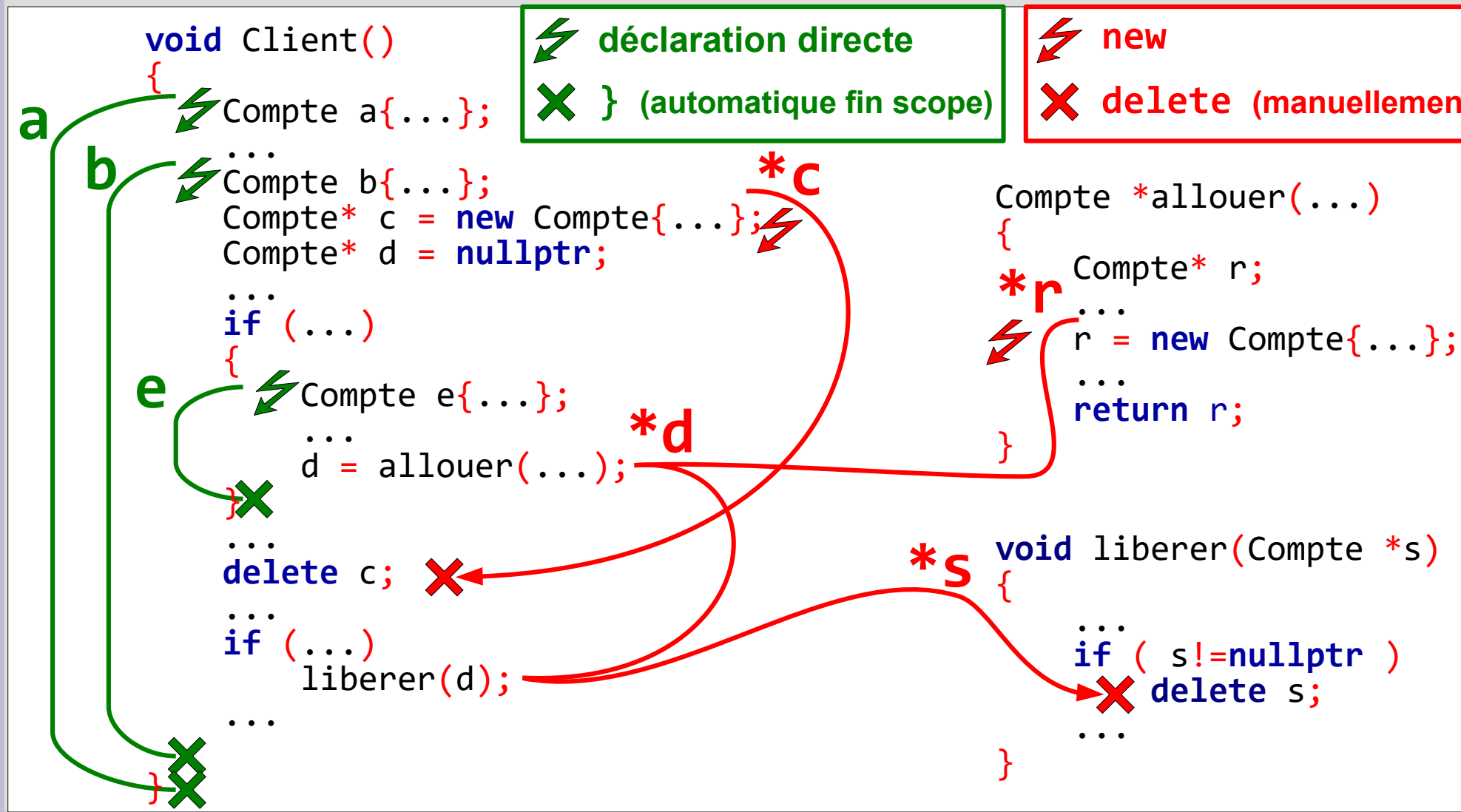
Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **dynamiques : entités persistantes***



Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction
*Les objets **automatiques** / **dynamiques***



Cycles de vie des objets

Cycle de vie objet : création-utilisation-destruction

- *Les objets **automatiques** : simples, efficaces, locaux, sécures*
- *Les objets **dynamiques** : situations complexes, persistants, risqués*
- *Les **entités** sont les objets persistants du modèle*
  La nature imprévisible du cycle de vie des entités conduit à utiliser l'allocation dynamique...
- *C'est une énorme problématique : 2 approches*

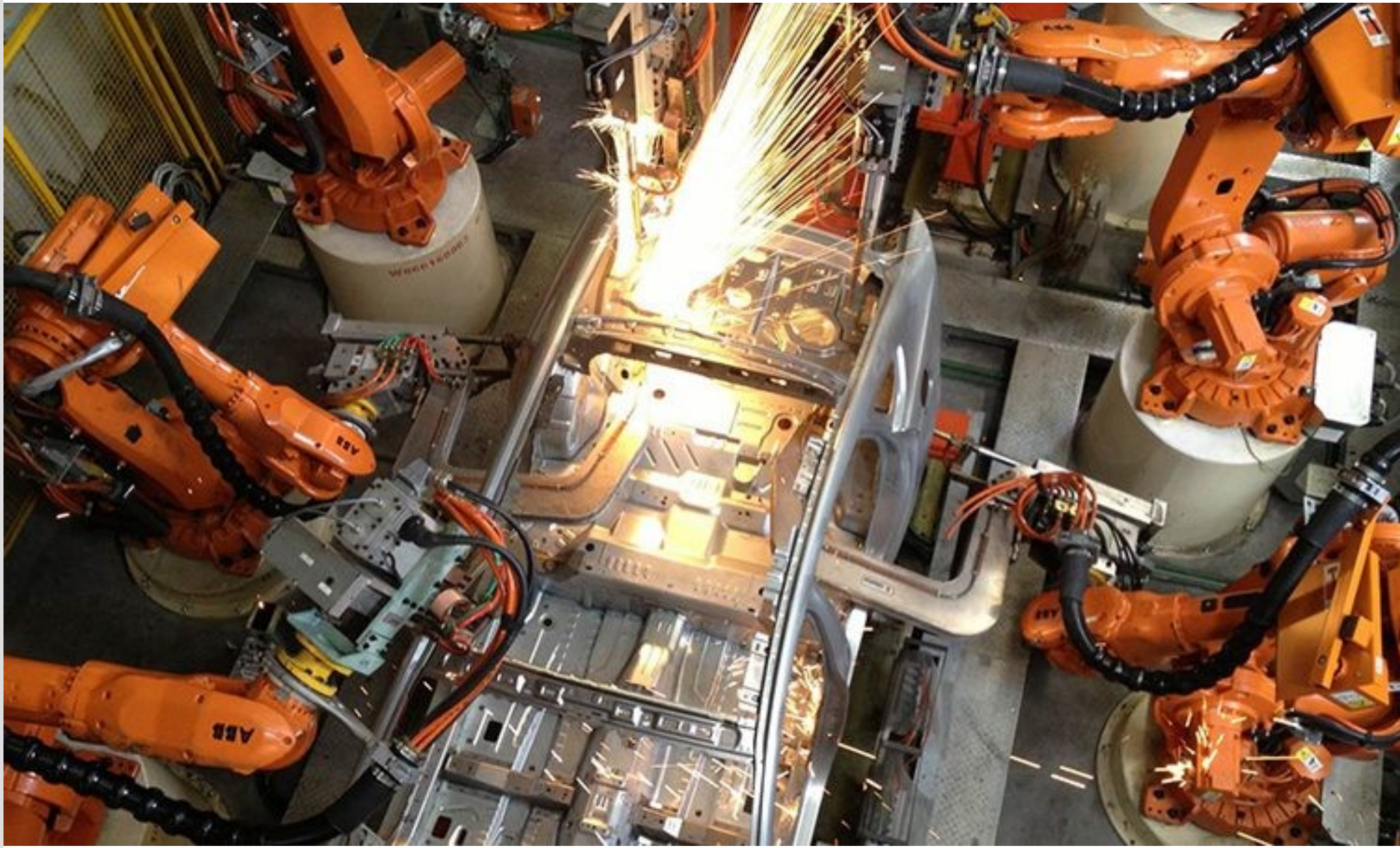
➔ ***Garbage collector** : Java, C#... pas de delete !*

➔ ***C++** objets spécialisés gestion de ressources...*

COURS 5

A) La classe en C++
B) L'encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Constructeur(s)



Constructeur(s)

*Un **constructeur** est une méthode spéciale*
*Appelée à la **création** de l'objet pour l'**initialiser***

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        ...
};
```

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
{
    m_titulaire = titulaire;
    m_solde     = solde_init;
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    Compte b{"Zag"};
```

Constructeur(s)

*Un constructeur a le **même nom** que sa classe*
*Il n'a **pas de type de retour**, même pas void !*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
      Compte(std::string titulaire, float solde_init=0);
        ...
};
```

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
{
    m_titulaire = titulaire;
    m_solde     = solde_init;
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    Compte b{"Zag"};
```

Constructeur(s)

*Un **constructeur** ne **réserve pas la mémoire** de l'objet créé : la zone mémoire pointée par* `this` *existe déjà au début de l'exécution du constructeur*

Action générée implicitement par le compilateur
*Allocation **automatique** de la mémoire objet*

```
this = malloc(1*sizeof(Compte))
```

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
{
    m_titulaire = titulaire;
    m_solde     = solde_init;
}
```

Pas d'allocation dans un constructeur
On initialise les valeurs des attributs

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
```

Constructeur(s)

*Le but du constructeur est de livrer un objet dans un **état** utilisable par le code client, en général on souhaite que tous les attributs soient initialisés*

On peut également faire d'autres actions, des calculs, des saisies, des ouvertures de fichier...

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
{
    std::cout << "Creation du compte " << titulaire << std::endl;

    if ( solde_init < 0 ) // Est-ce une bonne idée de vouloir
        solde_init = 0;   // "rattraper le coup" ? => signaler pb.

    m_titulaire = titulaire;
    m_solde     = solde_init;
}
```

Constructeur(s)

Il est fréquent que le constructeur commence par copier directement les valeurs des paramètres dans les attributs : dans ce cas il est ***préférable*** *d'utiliser la syntaxe* ***spécifique aux constructeurs*** *dite* ***liste d'initialisation*** *(* `≠ std::initializer_list` *)*

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
    : m_titulaire{titulaire}, m_solde{solde_init}
{
    std::cout << "Creation du compte " << titulaire << std::endl;

    if ( m_solde < 0 ) // Est-ce une bonne idée de vouloir
        m_solde = 0;   // "rattraper le coup" ? => signaler pb.
}
```

Constructeur(s)

Souvent on n'a pas d'autres actions à faire que d'initialiser les attributs : dans ce cas le corps de la méthode reste vide !

compte.cpp

```
Compte::Compte(std::string titulaire, float solde_init)
    : m_titulaire{titulaire}, m_solde{solde_init}
{ }
```

*Le std::string m_titulaire est directement créé avec sa valeur finale : **plus efficace***

Le corps d'un constructeur avec liste d'initialisation reste souvent vide (il faut quand même le mettre)

Constructeur(s)

*Le **constructeur par défaut** est le constructeur qui sait construire l'objet avec aucun paramètre*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        ...
};
```

Classe sans constructeur par défaut

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    Compte b{"Zag"};
    Compte c{};          error: no matching function
    Compte c;            for call to 'Compte::Compte()'
```

idem

Constructeur(s)

*On obtient un **constructeur par défaut** avec des valeurs par défaut de paramètres d'un constructeur*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire="", float solde_init=0);
        ...
};
```

Classe avec constructeur par défaut

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    Compte b{"Zag"};
idem Compte c{};
     Compte c;
```

Ok c sera un objet initialisé par défaut

Constructeur(s)

*On obtient un **constructeur par défaut** avec un constructeur sans paramètre*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        Compte();
};
```

Surcharge de constructeur !

constructeur avec paramètres

constructeur par défaut

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    Compte b{"Zag"};
idem Compte c{};
     Compte c;
```

Ok c sera un objet initialisé par le constructeur par défaut

Constructeur(s)

Le constructeur par défaut peut se contenter de mettre des valeurs « neutres »

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        Compte();
};
```

compte.cpp

```
Compte::Compte()
    : m_titulaire{""}, m_solde{0}
{ }
```

client.cpp

```
void client()
{
    Compte x;
    Compte y;
```

Ok x et y seront des objets initialisés par le constructeur par défaut

Constructeur(s)

Le constructeur par défaut peut si ça fait sens mettre des valeurs spécifiques

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        Compte();
};
```

compte.cpp

```
Compte::Compte()
    : m_titulaire{"AUCUN"}, m_solde{0}
{ }
```

client.cpp

```
void client()
{
    Compte x;
    Compte y;
}
```

Ok x et y seront des objets initialisés par le constructeur par défaut

Constructeur(s)

*Un constructeur peut **déléguer** le travail d'initialiser à un autre constructeur...*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        Compte();
};
```

compte.cpp

```
Compte::Compte()
    : Compte{"AUCUN"}
{ }
```

client.cpp

```
void client()
{
    Compte x;
    Compte y;
}
```

Ok x et y seront des objets initialisés par le constructeur par défaut

Constructeur(s)

*Un constructeur par défaut peut si ça fait sens aller **récupérer ailleurs les valeurs initiales***

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        Compte();
};
```

compte.cpp

```
Compte::Compte()
{
    std::cin >> m_titulaire;
    std::cin >> m_solde;
}
```

client.cpp

```
void client()
{
    Compte x;
    Compte y;
}
```

Ok x et y seront des objets saisis par l'utilisateur !

Constructeur(s)

En l'absence de tout constructeur déclaré le compilateur en fournit un par défaut implicite...

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        // PAS DE Compte( ) DÉCLARÉ !
        // Il peut y avoir d'autres méthodes...
};
```

Mieux vaut ne pas se fier au constructeur par défaut fourni implicitement

compte.cpp

```
// PAS DE Compte::Compte( ) DÉFINI.
```

client.cpp

```
void client()
{
    Compte a;       // m_solde = PAS INITIALISÉ
```

Méfiance

Constructeur(s)

On n'est pas obligé d'avoir un constr. par défaut
*Il est quasi-obligatoire d'avoir **au moins** un constr.*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire);
        ...// Pas d'autre constructeur
};
```

compte.cpp

```
Compte::Compte(std::string titulaire)
    : m_titulaire{titulaire}, m_solde{0}
{ }
```

client.cpp

```
void client()
{
    Compte a{"Zig"};
    Compte b{"Zag"};
}
```

Ok une seule façon de créer un objet Compte

Constructeur(s)

Pour qu'un objet fonctionne « par valeur » (copies) le compilateur génère des méthodes implicites...

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire);
        ...// Pas d'autre constructeur
};
```

client.cpp

```
void client()
{
    Compte a{"Zig"};
    Compte b{"Zag"};
    Compte c{a};
    Compte d = b;
    c = b;
    b = Compte("Zog");
    a = Compte{"Zug"};
    d = {"Zyg"};
}
```

Ces méthodes implicites copient un par un les attributs de l'objet source vers l'objet destination
On peut les coder explicitement (on verra plus tard)

Ok constructeur par copie implicite

Ok opérateur d'affectation implicite
(3 derniers : on passe par un objet temporaire)

Constructeur(s)

Beaucoup (trop) de façons de déclarer un objet !
La forme parenthèses est encore très courante

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire="", float solde_init=0);
        ...// Pas d'autre constructeur
};
```

client.cpp

```
void client()
{
    Compte a{"Zig"};
    Compte b("Zag");

    Compte c;
    Compte d{};
    Compte e();
```

braced initialization (C++11)

Initialisation traditionnelle (comme un appel)

Objet par défaut ok

Déclaration d'une fonction e retournant un objet Compte

Constructeur(s)

Beaucoup (trop) de façons de déclarer un objet !

Ces variantes sont là pour des raisons techniques (compatibilité, cohérence) et pratiques (lisibilité)

Ne soyez pas surpris de voir des déclarations bizarroïdes dans les exemples sur les forums y compris avec de simples types scalaires...

```
for (int i=0; i<10; ++i) { ... } // Classique et de bon goût

for (int i(0); i<10; ++i) { ... } // Très tendance... en 2017

for (int i{0}; i<10; ++i) { ... } // Printemps-été 2020 ?

for (int i={0}; i<10; ++i) { ... } // Abusé ! (ça marche)
```

COURS 5

A) La classe en C++
B) L’encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Destructeur

Destructeur

*Le **destructeur** est une méthode spéciale*
*Appelée à la **destruction** de l'objet, pour quoi faire ?*

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
};
```

compte.cpp

```
Compte::~Compte()
{
    // Rien !
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

Destructeur

*Le destructeur a le **nom** de sa classe préfixé par **~***
Ni type de retour ni paramètre**, il est **unique

compte.h

```
class Compte
{   private :
        std::string m_titulaire;
        float       m_solde;

    public :
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();
};
```

compte.cpp

```
Compte::~Compte()
{
    // Rien !
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

Destructeur

*Le **destructeur** ne **libère pas la mémoire** de l'objet détruit : la zone mémoire pointée par* `this` *existe jusqu'à la fin du code du destructeur*

L'objet est donc pleinement utilisable dans le destructeur : par exemple on peut l'afficher ...

Pas de libération dans un destructeur

compte.cpp

```
Compte::~Compte()
{
    // Rien !
}
```

Action générée implicitement par le compilateur
*Libération **automatique** de la mémoire objet*

```
Appel destructeur chaque attributs puis
free(this)
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

Destructeur

*Le **destructeur** ne **libère pas la mémoire** de l'objet détruit : la zone mémoire pointée par* `this` *existe jusqu'à la fin du code du destructeur*

L'objet est donc pleinement utilisable dans le destructeur : par exemple on peut l'afficher ...

Pas de libération dans un destructeur

compte.cpp

```
Compte::~Compte()
{
   std::cout << "Liberation du compte " << m_titulaire <<std::endl;
}
```

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

Destructeur

Le but du destructeur est de libérer des ressources (autres que lui même) qui auraient été ***acquises*** *à sa création ou durant son utilisation*

Typiquement un attribut de l'objet pointe sur une ressource dynamique (obtenue avec new) et l'objet a la responsabilité de le libérer (faire delete) C'est un sujet délicat (plus tard...)

compte.cpp

```
Compte::~Compte()
{
    // Rien pour l'instant car pas de new dans le constructeur !
}
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

Destructeur

En effet pour l'instant le destructeur ne nous sert pas à grand chose à part éventuellement afficher ou compter les destructions d'objet.

Si ces actions ne nous intéressent pas, le destructeur implicite généré par le compilateur convient parfaitement (contrairement au constr.)

compte.h

```
class Compte
{
        // PAS DE ~Compte( ) DÉCLARÉ !
        // Il peut y avoir d'autres méthodes, constructeur(s)...
};
```

compte.cpp

```
// PAS DE ~Compte::Compte( ) DÉFINI.
```

client.cpp

```
void client()
{
    Compte a{"Zig", 100};
    ... // utiliser l'objet a
}
```

OK l'objet a est bien détruit
Même si on n'a pas écrit de destructeur explicitement

COURS 5

A) La classe en C++
B) L'encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Accesseurs et mutateurs

Accesseurs et mutateurs

Un code client mécontent :

« Comment ? J'ai sous les yeux un objet Compte que j'ai rempli moi même à l'instant, et vous me dite que je ne peux pas savoir qui est son titulaire ?! »

```
error: 'std::__cxx11::string Compte::m_titulaire' is private
error: within this context
```

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    std::cout << monCompte.m_titulaire << std::endl;

}
```

Accesseurs et mutateurs

Au début on a l'impression que la POO n'est faite que de restrictions ni très drôles ni très productives

*Bien sûr **si** connaître le titulaire d'un compte est un **besoin légitime** du code client alors le concepteur de la classe a prévu un **accesseur** `getTitulaire` mais pas de rendre publique l'attribut m_titulaire...*

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    std::cout << monCompte.getTitulaire() << std::endl;

}
```

OK, Compte offre un accesseur en lecture publique

Accesseurs et mutateurs

- *Le travail du concepteur de la classe consiste à fournir une interface cohérente vis-à-vis du **rôle** que vont jouer les objets de la classe et des **comportements** qu'on en attend*
- *Avoir des points d'accès aux attributs privés est **souvent** un comportement souhaité*
- *Une méthode qui permet d'accéder en **lecture** à une donnée privée est un **accesseur** ou **accesseur en lecture** ou **getter***
- *Une méthode qui permet d'accéder en **écriture** à une donnée privée est un **mutateur** ou **accesseur en écriture** ou **setter***

Accesseurs et mutateurs

- *Par convention getters et setters sont préfixés par* **`get`** *ou* **`set`**

- ***Accesseur*** *ou* ***getter*** *d'un attribut* **`m_solde`** *:*

```
float getSolde() const;
```

ou

```
float get_solde() const;
```

Noter qu'un getter est en principe en lecture seule donc il ne doit pas modifier les données de l'objet => const

- ***Mutateur*** *ou* ***setter*** *d'un attribut* **`m_solde`** *:*

```
void setSolde(float nouveauSolde);
```

ou

```
void set_solde(float nouveauSolde);
```

Accesseurs et mutateurs

*Une **mauvaise** compréhension de l'orienté objet : tous les attributs en privé, mais pour tous un accesseur et un mutateur publique*

compte.h

```
class Compte
{
    private :
        /// Attributs
        std::string m_titulaire;
        float       m_solde;

    public :
        /// Constructeur(s), Destructeur
        ...
        /// Accesseurs et mutateurs
        std::string getTitulaire() const;
        void setTitulaire(std::string nouveauTitulaire);

        float getSolde() const;
        void setSolde(float nouveauSolde);
};
```

Accesseurs et mutateurs

*Une **mauvaise** compréhension de l'orienté objet : tous les attributs en privé, mais pour tous un accesseur et un mutateur publique*

compte.h

Comme faire des trous dans un sous-marin !

```
class Compte
{
    private :
        /// Attributs
        std::string m_titulaire;
        float       m_solde;

    public :
        /// Constructeur(s), Destructeur
        ...
        /// Accesseurs et mutateurs
        std::string getTitulaire() const;
        void setTitulaire(std::string nouveauTitulaire);

        float getSolde() const;
        void setSolde(float nouveauSolde);
};
```

Accesseurs et mutateurs

*Une **mauvaise** compréhension de l'orienté objet : tous les attributs en privé, mais pour tous un accesseur et un mutateur publique*

compte.cpp

```
std::string Compte::getTitulaire() const
{
    return m_titulaire;
}

void Compte::setTitulaire(std::string nouveauTitulaire)
{
    m_titulaire = nouveauTitulaire;
}

float Compte::getSolde() const
{
    return m_solde;
}

void Compte::setSolde(float nouveauSolde)
{
    if ( nouveauSolde>=0 )
        m_solde = nouveauSolde;
    /// else what ?
}
```

Dans un setter on peut en profiter pour faire du contrôle des valeurs

Mais l'objet est-il en situation de décider ce qu'il convient de faire ?

Accesseurs et mutateurs

*Une **mauvaise** compréhension de l'orienté objet : pour chaque attribut accesseur / mutateur publique au final on utilise la classe comme une struct !*

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    /// Débiter 50 à M. Durand
    float aDebiter = 50;
    monCompte.m_solde -= aDebiter;
```

```
error: 'float Compte::m_solde'
       is private
error: within this context
```

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    /// Débiter 50 à M. Durand
    float aDebiter = 50;
    monCompte.setSolde( monCompte.getSolde() - aDebiter );
```

Avec les accesseurs ça passe...
*Mais c'est un peu **lourd** non ?*
C'est ça la programmation objet ?

Accesseurs et mutateurs

*Une **mauvaise** compréhension de l'orienté objet : pour chaque attribut accesseur / mutateur publique au final on utilise la classe comme une struct !*

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    /// Débiter 50 à M. Durand
    float aDebiter = 50;

    if ( aDebiter <= monCompte.getSolde() )
        monCompte.setSolde( monCompte.getSolde() - aDebiter );
    else
    {
        std::cout << "échec de débit " << aDebiter;
        /// Traiter l'échec ...
    }
```

Avec les accesseurs ça passe...

Mais c'est un peu lourd non ?

C'est ça la programmation objet ?

Accesseurs et mutateurs

*Une **bonne** compréhension de l'orienté objet : on a juste les accesseurs / mutateurs nécessaires pour les **comportements** attendus de l'objet*

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    /// Débiter 50 à M. Durand
    float aDebiter = 50;

    if ( monCompte.solvable(aDebiter) )
        monCompte.debiter(aDebiter);
    else
    {
        std::cout << "échec de débit " << aDebiter;
        /// Traiter l'échec ...
    }
```

Avec les bons accesseurs ça passe...
Et ça s'utilise sans trop de lourdeur.

Accesseurs et mutateurs

*Une **bonne** compréhension de l'orienté objet :*
on a juste les accesseurs / mutateurs nécessaires
*pour les **comportements** attendus de l'objet*

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};

    /// Débiter 50 à M. Durand
    float aDebiter = 50;

    if ( monCompte.solvable(aDebiter) )
        monCompte.debiter(aDebiter);
    else
    {
        std::cout << "échec de débit " << aDebiter;
        /// Traiter l'échec ...
    }
```

Avec les bons accesseurs ça passe...
Et ça s'utilise sans trop de lourdeur.

Accesseurs et mutateurs

- *On a envie (ça se discute) d'appeler les méthodes* **`solvable`** *un accesseur et* **`debiter`** *un mutateur*
- *Peu importe : on comprend qu'il faut éviter que le code client gère lui même des « détails »*
- *Pas toujours si simple !*
- *Le code client doit* ***déléguer*** *les opérations à l'objet mais l'objet a une vision* ***locale*** *de ses données, seul le code client a une vision plus large du* ***contexte****, ça demande de l'expérience mais la plupart du temps on peut descendre les infos utiles en paramètres...*
- *Les « cas à problème » remontent à l'appelant*

Accesseurs et mutateurs

Une façon de « remonter à l'appelant » les cas à problème (on verra une alternative : les exceptions)

compte.cpp

```
bool Compte::debiter(float debit)
{
    if ( !solvable(debit) )
        return false;

    m_solde -= debit;
    return true;
}
```

client.cpp

```
void client()
{
    Compte monCompte{"Durand", 100};
    float aDebiter = 50;

    if ( !monCompte.debiter(aDebiter) );
    {
        std::cout << "échec de débit " << aDebiter;
        /// Traiter l'échec ...
    }
```

Accesseurs et mutateurs

En résumé : choisir soigneusement les méthodes !

compte.h

Ceci n'est pas un corrigé officiel : tout dépend du CDC !

```
class Compte
{
    private :
        /// Attributs privés
        std::string m_titulaire;
        float m_solde;

    public :
        /// Méthodes publiques (interface)
        Compte(std::string titulaire, float solde_init=0);
        ~Compte();

        std::string getTitulaire() const;
        float getSolde() const;

        bool solvable(float montant) const;
        void crediter(float credit);
        bool debiter(float debit);
        void afficher() const;
};
// Ceci n'est pas une méthode mais une fonction associée à la classe
void transferer(Compte& debiteur, Compte& beneficiaire, float montant);
```

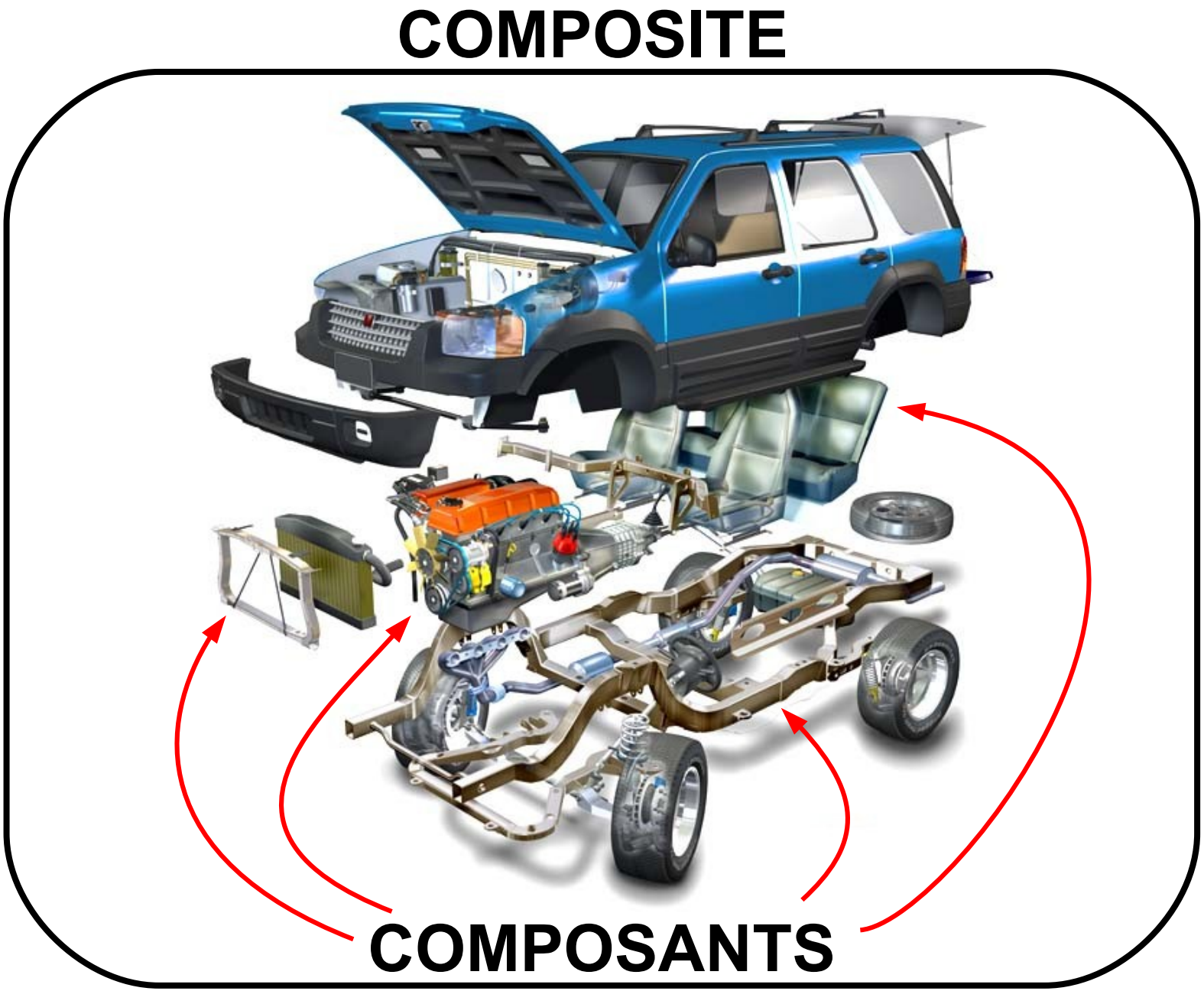
Ici on a choisi de ne pas avoir de mutateur de m_titulaire : on considère que cette valeur doit être invariante dès la création de l'objet

Le solde est modifiable mais pas directement ...

COURS 5

A) La classe en C++
B) L'encapsulation
C) Les méthodes, this, const
D) Cycles de vie des objets
E) Constructeur(s)
F) Destructeur
G) Accesseurs et mutateurs
H) Composition entre classes

Composition entre classes

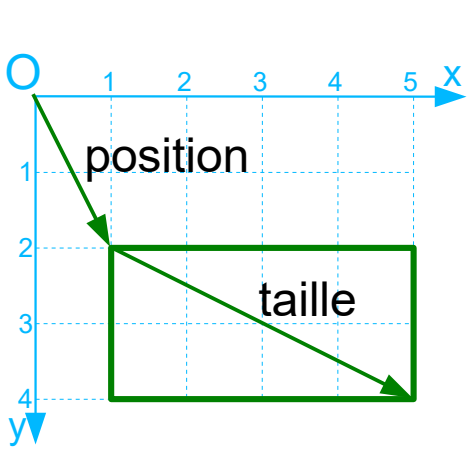


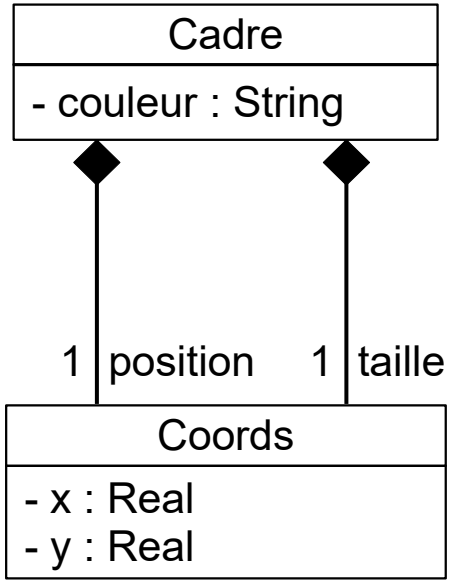
Composition entre classes

- *On a déjà composé une classe dans une classe puisqu'on a utilisé un attribut std::string titulaire dans la classe Compte et...*
- *std::string est une classe !*
- *C'est une classe bien conçue : l'utiliser comme type attribut ne pose aucun problème*
- *On peut bien sûr composer avec nos propre classes selon nos besoins*
- *Avec les « objets-valeurs » ça se passera bien : objets qui ne dépendent de personne, non partagés (sémantique par valeur) et copiables*

Composition entre classes

- *On a déjà parlé d'une struct **Coords**, on peut la transformer en classe et l'utiliser comme composant d'un nouveau type **Cadre** :*





cadre.h

```
class Cadre
{
    private :
        Coords m_position;
        Coords m_taille;
        std::string m_couleur;

    public :
        ...
};
```

coords.h

```
class Coords
{
    private :
        double m_x;
        double m_y;

    public :
        ...
};
```

Composition entre classes

- *Utiliser une classe comme composant implique souvent de faire un peu de plomberie au niveau des constructeurs (il faut construire les attributs)*

cadre.cpp

```
Cadre::Cadre(double xpos, double ypos,
             double largeur, double hauteur,
             std::string couleur)
    : m_position{xpos, ypos},
      m_taille{largeur, hauteur},
      m_couleur{couleur}
{ }
```

coords.cpp

```
Coords::Coords(double x, double y)
    : m_x{x}, m_y{y}
{ }
```

Composition entre classes

- *Si la classe composante propose des opérations intéressantes le travaille de la classe composite est simplifié (ici : calcul vectoriel direct)*

cadre.cpp

```
void Cadre::calerBasDroite(const Cadre& parent)
{
    m_position = parent.m_position + parent.m_taille - m_taille;
}
```

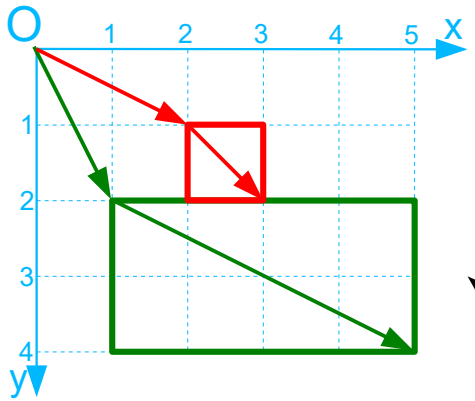
coords.cpp

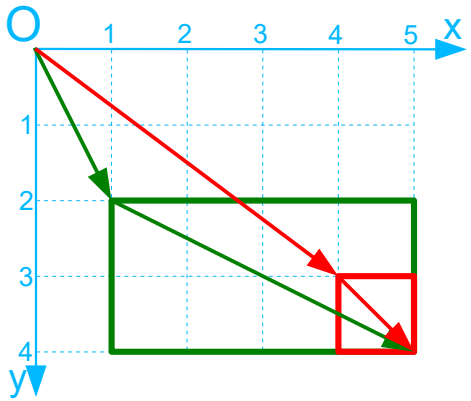
```
Coords operator+(const Coords& a, const Coords& b)
{
    return {a.m_x + b.m_x, a.m_y + b.m_y };
}

Coords operator-(const Coords& a, const Coords& b)
{
    return {a.m_x - b.m_x, a.m_y - b.m_y };
}
```

Composition entre classes

- *Le client de la classe composite profite de fonctionnalités avec un haut niveau d'abstraction*






```
position=(4, 3)
taille=(1, 1)
couleur=red
```

client.cpp

```
void client()
{
    Cadre grand(1, 2, 4, 2, "green");
    Cadre petit(2, 1, 1, 1, "red");

    petit.calerBasDroite(grand);
    petit.afficher();
}
```

cadre.cpp

```
void Cadre::calerBasDroite(const Cadre& parent)
{
    m_position = parent.m_position + parent.m_taille - m_taille;
}
```