



ISIE4 – LOO C++

Classes et Objets

Une classe / Un objet

- Une classe est une définition d'un nouveau type ***concret***

```
class MaClasse {  
    ...  
};
```

- MaClasse est un nouveau type
 - On peut déclarer/définir des variables de ce type
 - Ce qui permet d'obtenir des objets (que l'on qualifiera de construits automatiquement automatic storage)

```
int unEntier; // un nouvel entier (définition)  
MaClasse unObjet; // un nouvel objet à moi...
```

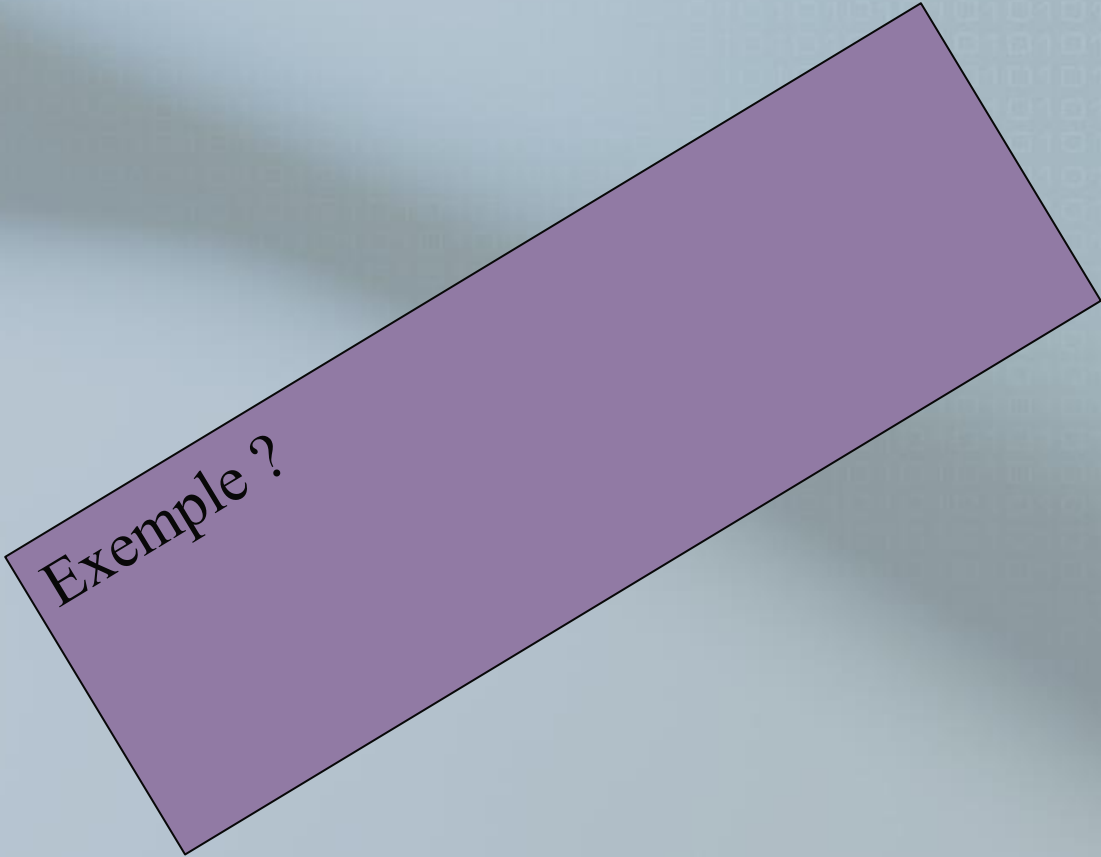
Attributs / Comportements

Champs / Méthodes

- Une classe est un type
 - Il faut lui associer des valeurs et des opérations (attributs et comportements)
- Éléments constitutifs de la déclaration d'une classe:
 - Champs (attributs)
 - Variables d'instance (propres à chaque objet de la classe)
 - Méthodes (comportements)
 - Opérations à appliquer aux objets de la classe (fonctions attachées aux objets)

Déclaration

- Suffit à imaginer une utilisation possible d'un objet de la classe



Exemple ?

Définition

- Définition des méthodes
 - Fournir le code
 - Utilisation de l'opérateur de portée `::` pour désigner la méthode

Exemple ?



Fonction != Méthode

- Fonction
 - Code nommé et paramétré
- Méthode (ou fonction membre, sic):
 - Réalisation concrète d'une opération d'un objet
- En C++ les définitions des méthodes et des fonctions sont syntaxiquement similaires, mais c'est sémantiquement très différent.
 - La ressemblance ne doit pas être trompeuse



Structure code source

- Déclaration d'une classe
 - Dans un fichier d'entête
 - ***MaClasse.hpp***, ***MaClasse.H*** ou ***MaClasse.h***
- Définition d'une classe
 - De ses méthodes en réalité
 - Dans un fichier ***MaClasse.cpp*** ou ***MaClasse.C***
 - Ou, pour certaines, directement dans la définition de classe (***.hpp***)
- L'utilisation de la classe nécessite l'import du fichier de déclaration via la directive **`#include`**

Initialisation des objets

- Si un objet est mal initialisé, c'est parce qu'à sa construction son fabricant s'y est mal pris !
 - Vrai dans la vie, doit être vrai en programmation.
- Ce n'est pas une solution que de proposer une méthode réalisant une « pseudo » initialisation
 - L'utilisateur peut être tenté de l'utiliser à des moments peu opportuns.



Solution

- On propose alors de contrôler l'initialisation des objets (rappel l'initialisation ne se produit qu'une fois dans la vie d'une variable : à sa création) :
 - En garantissant des valeurs par défaut cohérentes
 - En obligeant l'utilisateur à fournir des valeurs le permettant
- En C++, il existe la possibilité de définir des méthodes très spéciales permettant de contribuer à l'initialisation d'un objet :
 - Les constructeurs (***ctors***)
 - Attention cependant : tout ce qui peut être initialisé sans l'intervention d'un constructeur doit l'être.

Les constructeurs (ctors)

- Méthodes
- ne portent pas de nom
- Déclarent le type de la classe comme type de retour
- Mais ne renvoient rien (return seul est autorisé).
- Sont appelées automatiquement après la création de l'objet (appel implicite) et ne peuvent être appelées explicitement.

Surcharge de constructeur

- Il est fréquent et ordinaire que plusieurs ctors soient définis.
 - Chacun d'eux correspond pour l'utilisateur à la façon de commander/obtenir un objet.

Attention : ctor sans paramètres

- MaClasse MonObjet;
 - Ctor sans paramètres, instantiation classique
- Mieux :
 - ***Maclasse MonObjet{}***
 - Réalise une zero-init des membres de l'objet créé
- MaClasse MonObjet();
 - Autre chose
 - Déclaration d'une fonction ne prenant pas de paramètres (ou un nombre de paramètres inconnu en C) et retournant un « MaClasse »

```
int main() {  
    int f();  
}  
vexing parse
```

Delegating ctor

- Appel croisé de constructeurs (comme en JAVA paraît-il...)
 - Delegating ctor / Delegated ctor
 - Autorisé depuis C++11
- Très utile pour rassembler toutes les opérations communes à chacun des constructeurs
 - Règle do DRY
 - Don't Repeat Yourself

Responsabilités

- Clairement définies
- L'utilisateur est obligé de se conformer aux scénarios de constructeur d'un objet
 - Si cela ne construit pas un objet correct ce n'est pas de sa faute!
 - En cas de mauvais comportement de l'objet, il peut se plaindre au fabricant (bien identifié) qui ne peut se défaire de sa responsabilité...
- Le concepteur est responsable:
 - De la bonne initialisation des attributs d'un objet, de sorte que celui-ci se comporte correctement
 - De la cohérence des données (internes)

Pause syntaxe...

- Rappel : pour les types primitifs, on a

```
int i=4;    // l'initialisation habituelle  
int j(4);   // l'initialisation fonctionnelle  
int k{4};   // l'initialisation C++11
```

- Pour les objets on a :

```
MaClasse c(12, "Bonjour", Math::sin(Math::PI));  
MaClasse d{12, "Bonjour", Math::sin(Math::PI)}; // recommandé C++11
```

- Si, pour une classe donnée, il existe un constructeur à un argument alors les syntaxes autorisées sont:

```
MyClass o{12};  
MyClass o(12);    // l'initialisation standard vue comme fonction  
MyClass o = 12;   // l'initialisation comme les types primitifs...
```



de l'uniformité des initialisations C++11

■ Pour les objets

```
MyClass o(12);    // l'initialisation vue comme appel d'1 ctor  
MyClass o = 12;   // l'initialisation comme les types primitifs...  
MyClass o{12};    // l'initialisation C++11 (short style)  
MyClass o = {12}; // l'initialisation C++11 (verbose style)
```

■ Pour les types primitifs

```
int o(12);        // type primitif vu comme une classe  
int o = 12;       // l'initialisation à la C...  
int o{12};        // l'initialisation C++11  
int o = {12};     // l'initialisation C++11
```

■ Préférer les initialisations C++11

■ short ou verbose



Ressources externes

- Les objets sont parfois employés comme interfaces d'accès à des ressources externes
 - Un objet détient une référence sur la ressource
 - Ne la contient pas
- Les constructeurs sont employés pour initier/initialiser cette relation

Exemple

■ Classe d'objets représentant une pile

```
class Pile {  
private:  
    int *stock;  
    int tailleMax, sommet;  
public:  
    Pile(int tailleMax);  
};  
  
Pile::Pile(int tm) {  
    stock = new int[tm]; // allocation « externe »  
    tailleMax = tm;  
    sommet = 0;  
}
```

Question

- Que se passe-t-il lors de la disparition de tels objets ?
- Que devient la relation établie ?
- Comment éviter ces fuites mémoires ?
 - Pas de garbage collector en C++

La destruction par destructeur (dtor)

- Un destructeur (dtor)
- Une méthode dont le nom est celui de la classe préfixée par le caractère ~
- Une méthode ne déclarant rien à renvoyer (rien pas void) et ne prenant pas de paramètre
- Une méthode automatiquement appelée à la destruction d'un objet. Cette destruction est implicite dans le cas d'un objet statiquement alloué ou explicite lorsqu'il s'agit de détruire un objet alloué dynamiquement
- Une méthode qui ne peut être surchargée.
 - Un seul destructeur par classe
- Attention il faut la qualifier de ***virtual*** (on verra plus tard pourquoi mais c'est essentiel)



Retour à l'exemple « pile »

```
class Pile {
private:
    int *stock;
    int tailleMax, sommet;
public:
    Pile(int tailleMax);
    virtual ~Pile();
};

Pile::Pile(int tm) {
    stock = new int[tm];
    tailleMax = tm;
    sommet = 0;
}

Pile::~~Pile() { // plus de fuite mémoire!!!!
    delete [] stock;
}
```

Ctor & Dtor

```
1  #ifndef __POUET_H__
2  #define __POUET_H__
3  #include <iostream>
4
5  class Pouet{
6  public:
7      Pouet();
8      virtual ~Pouet();
9
10 };
11 #endif
12
```

```
#include <iostream>

#include "pouet.hpp"

void Dummy(Pouet poua, Pouet poub) {
    Pouet *pPouc = new Pouet();
    delete pPouc;
}

int main() {
    Pouet pa, pb;

    Dummy(pa, pb);

    return 0;
}
```

```
2  #include "pouet.hpp"
3
4
5  Pouet::Pouet()
6  {
7      std::cout<<"Construction d'un Pouet."<<std::endl;
8  }
9
10 Pouet::~~Pouet()
11 {
12     std::cout<<"Destruction d'un Pouet."<<std::endl;
13 }
14
15
```


Résultat de l'exécution

```
alexis@ubuntu-AXR: ~/LOO_CPP/Exemples/ConstructionDestruction/build 88x20
alexis@ubuntu-AXR: ~/LOO_CPP/Exemples/ConstructionDestruction/build$ ./CtorDtor_xple
Construction d'un Pouet.
Construction d'un Pouet.
Construction d'un Pouet.
Destruction d'un Pouet.
Destruction d'un Pouet.
Destruction d'un Pouet.
Destruction d'un Pouet.
Destruction d'un Pouet.
alexis@ubuntu-AXR: ~/LOO_CPP/Exemples/ConstructionDestruction/build$
```

- On observe
 - 3 exécutions de constructeurs
 - 5 exécutions de destructeurs
- ???

Explication – Le début

■ Rappel

- Le passage d'argument du C++ est un passage par valeur (comme en C)

■ Cela veut dire

- À l'entrée d'une fonction recevant un argument par valeur, on crée une nouvelle variable du type considéré et dont le nom est celui du paramètre formel, variable qui est alors initialisée à l'aide de la valeur du paramètre effectif



Explication – Le milieu

- Dans l'exemple on crée deux objets « Pouet » à l'entrée de la fonction
 - On ne voit pas l'appel du ctor
 - ???
- Il existe donc une notion de construction d'un objet de type T à l'aide d'un objet de type T (une sorte de clonage).
 - Ce mécanisme est appelé construction par copie (*copy ctor*)
 - Signature (imaginée d'un copy ctor) :
 - Pouet (Pouet);

Explication – la fin

- La bonne signature
 - Pouet(const Pouet&);
 - const pas obligatoire, mesure de prudence
- Passage de paramètre par référence constante
 - On ne crée pas de nouvelle variable mais on désigne dans la fonction le paramètre effectif à l'aide d'un autre nom (comme un alias)



Résultat

```
class Pouet{  
public:  
    Pouet();  
    Pouet(const Pouet&);    // Copy Ctor  
    virtual ~Pouet();  
};  
#endif
```

```
#include "pouet.hpp"  
  
Pouet::Pouet()  
{  
    std::cout<<"Construction d'un Pouet."<<std::endl;  
}  
  
Pouet::Pouet(const Pouet &aCloner)  
{  
    std::cout<<"Construction d'un Pouet (par copie)."<<std::endl;  
}  
  
Pouet::~~Pouet()  
{  
    std::cout<<"Destruction d'un Pouet."<<std::endl;  
}
```

main.cpp x pouet.cpp x pouet.hpp x

```
1 #include <iostream>  
2  
3 #include "pouet.hpp"  
4  
5 void Dummy(Pouet poua, Pouet poub) {  
6     Pouet *pPouc = new Pouet();  
7     delete pPouc;  
8 }  
9  
10 int main() {  
11     Pouet pa, pb;  
12  
13     Dummy(pa, pb);  
14  
15     return 0;  
16 }  
17
```

```
alexis@ubuntu-AXR:~/L00_CPP/Exemples/ConstructionDestruction_2/build$ ./CtorDtor_xple  
Construction d'un Pouet.  
Construction d'un Pouet.  
Construction d'un Pouet (par copie).  
Construction d'un Pouet (par copie).  
Construction d'un Pouet.  
Destruction d'un Pouet.  
Destruction d'un Pouet.  
Destruction d'un Pouet.  
Destruction d'un Pouet.  
Destruction d'un Pouet.  
alexis@ubuntu-AXR:~/L00_CPP/Exemples/ConstructionDestruction_2/build$
```

Précisons cette histoire de références...

- Que se passe-t-il si on souhaite que la fonction Dummy retourne un Pouet ?
 - Pouet Dummy(Pouet poua,Pouet poub);
- Que se passe-t-il si on passe les objets par référence ?
 - Pouet Dummy(Pouet &poua,Pouet &poub);
- Que se passe-t-il si on retourne l'objet par référence ?
 - Pouet &Dummy(Pouet &poua,Pouet &poub);

Précisons cette histoire de références... today...

- Aujourd'hui C++ exploite le « move-semantic »
 - `std::move`
 - Plus efficace que la copie
- The bad news
 - Nouveaux constructeurs (move ctor) et opérateur de « move »
- The good news
 - Avec les types « moveable »
 - La majorité des types std
 - Le compilateur exploite le move pour les valeurs retournées
 - On retourne directement l'objet 😊

Objets constants


- Définir des objets non-mutables
 - Constantes de l'ensemble défini
- Par exemple
 - Un point origine pour une classe Point
 - Une couleur « rouge » pour une classe Couleur
 - ...
- Il faut utiliser des objets ou types non mutables lorsque c'est possible
 - Meilleures optimisations

Variables constantes rappels

- les variables ***const*** sont des variables comme les autres, mais dont l'usage est restreint à un sous-ensemble des actions possibles.
 - Le sous-ensemble est défini comme étant l'ensemble des actions qui ne modifient pas la valeur.
 - Pour un type ordinaire: grossièrement pas d'affectation, ni incrémentation...

Pour une variable de type classe

- L'ensemble des actions accessibles sur un objet considéré comme constant est spécifié en ajoutant à chaque action l'attribut `const`



```
class Nombre {  
public:  
    Nombre(...);  
    long toLong() const;  
    int toInt() const;  
    void set(long v);  
    void set(int v);  
};
```

```
int main() {  
    Nombre n(...);  
    n.set(45L);  
    long l = n.toLong();  
    const Nombre zero(...);  
    l = zero.toLong();  
    zero.set(23);  
}
```

Attributs mutables

- La contrainte peut être trop forte...
On veut pouvoir modifier certains attributs d'un objet alors même qu'il est extérieurement (logiquement) considéré comme constant...
- Exemple : un compte en banque dont on imagine que les transactions soient toutes tracées et conservées en son sein, y compris les consultations... mais dont la valeur intrinsèque ne change pas... Des valeurs de cache, etc.

Attributs mutables

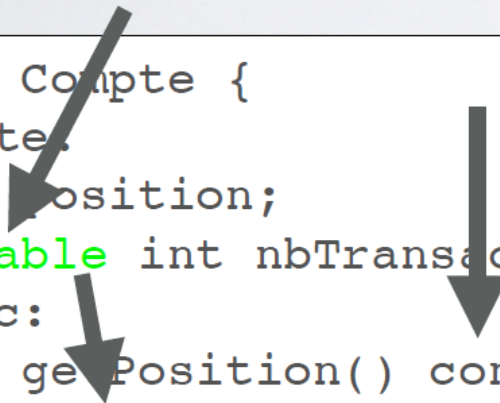
- dans le cas de la consultation (méthode `getPosition`), il est raisonnable de considérer cette méthode comme `const`
- Le problème est que certains attributs pourraient devoir être modifiés par la méthode `getPosition`
 - En ce cas, l'attribut peut être marqué comme mutable

De façon plus explicite...

```
class Compte {  
private:  
    int position;  
public:  
    int getPosition() const { return position; }  
};
```

```
class Compte {  
private:  
    int position;  
    int nbTransactions;  
public:  
    int getPosition() const {  
        nbTransactions++;  
        return position;  
    }  
};
```

Solution



```
class Compte {  
private:  
    int position;  
    mutable int nbTransactions;  
public:  
    int getPosition() const {  
        nbTransactions++;  
        return position;  
    }  
};
```


Attributs constants

- Il est naturel d'avoir à définir des attributs constants
 - Exemple : le numéro de sécurité sociale d'un individu
 - Dans ce cas, il suffit de rajouter à sa définition la qualification const
 - Comment les initialiser puisqu'aucune affectation n'est autorisée ?

```
class Individu{  
private:  
    const int numeroSecuriteSociale;  
};
```

Syntaxe d'initialisation spécifique – Niveau ctor

```
class Individu{
private:
    const int numeroSecuriteSociale;
public:
    Individu(...);
};

Individu::Individu(...) {
    numeroSecuriteSociale = ...;
}
```



```
class Individu{
private:
    const int numeroSecuriteSociale;
public:
    Individu(...);
};

Individu::Individu(...) : numeroSecuriteSociale(...)
{
}
```

this

- 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 *`



Levée d'ambiguïté

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

```
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);  
    }  
};
```

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)



Membres statiques

- 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

Déclaration / Utilisation

- Déclaration de donnée membre

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

- Initialisation

- Définition au niveau global

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

- Rq: C'est bien une variable globale (encapsulée)

Fonction membre statique

■ Déclaration

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

■ Définition

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

Remarques

- 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



Initialisation de membres statiques constants

- 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

- Membres statiques
 - Définition de constantes globales à la classe
 - Dénombrement d'instances, liste d'instances...
- Fonctions membres statiques
 - Accesseurs d'attributs statiques
 - Fonctions utilitaires (de service)

```
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 ?

Protection & contrôle d'accès

- Rappel : l'encapsulation
 - Boîte contenant des éléments
 - Classe comme structure de données et fonctions agissant sur celles-ci
 - Contrôle d'accès aux éléments
 - Mécanisme permettant de « cacher » certains mécanismes de fonctionnement
- Protection en domaines
 - Privé (par défaut)
 - Public

Domaine privé

- Mot réservé : private
- les éléments de ce domaine ne sont atteignables que depuis des fonctions membres (statiques ou non) de la classe.
- 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

Domaine privé – Classes non instanciables

- Constructeur en private
- Utilité ?
 - Classe « module »
 - Méthodes « static »
 - Design pattern ***Factory*** (usine)
 - Pour contrôler totalement le mécanisme de création d'un objet
 - Souvent via une méthode (statique) « create »
 - Design pattern ***Singleton***
 - Pour s'assurer que l'objet créé est réellement instancié une seule et unique fois



Domaine public

- Mot réservé : public
- les éléments de ce domaine sont atteignables depuis n'importe quel point du programme



Fonctions amies

- Il arrive que l'on souhaite pouvoir accéder, via des services spécifiques, non membres, à des données privées d'une instance
 - Sans utiliser d'accessseurs publics
- Ces services (fonctions) doivent être clairement identifiés/spécifiés et déclarés comme « amis »
 - friend



Remarques sur l'amitié

- Souvent utilisée dans le cadre de la surcharge d'opérateurs
 - Opérateurs d'arité 2 notamment
 - Inutile en réalité, si bonne conception
- Peut nécessiter dans certains cas une déclaration anticipée
 - forward declaration
- les déclarations d'amitiés peuvent apparaître dans n'importe quelle section, cela ne joue pas



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!!!

Initialisation des membres

- Mettre en place autant que faire se peut de vraies initialisations
 - Syntaxe C++11 préférable
 - On verra ça clairement en TP
- Ordre des créations & initialisations
 - Allocation de la mémoire
 - Pour tous les objets de la classe
 - Appel des constructeurs adéquats
 - Dans l'ordre de leur apparition dans la déclaration de la classe
 - Exécution du constructeur de la classe englobante
 - Les données membres étant donc déjà initialisées

Synthèse

- Classes : Déclaration & Définition
- Attributs & méthode
 - Accesseurs
- Ctor / Dtor
 - Surcharge de constructeur
 - Copy Ctor
- Objets constants / non mutables
 - Attributs mutables
- Membres statiques
 - Attributs & Méthodes
- Protection & Contrôle d'accès
- Fonctions amies
- Retour sur l'initialisation des membres