

MÉMO COURS N° 2 DE C++

- **Programmation Orientée Objet** = modèle d'abstraction.
- **Modélisation objet** : basée sur le concept de l'objet.
 - **Objet** = ensemble de données ou état (attributs = champs des données) ainsi que les opérations associées ou le comportement (quoi et comment) défini par les méthodes

Quoi ?	Comment ?
Modélisation	Implémentation
Point de vue utilisateur	Point de vue programmeur
Fonctions	Implémentation des fonctions

- Exemple : Un client bancaire est un objet. Il a un nom, prénom, numéro client... Les opérations associées sont par exemple consulter son solde, fermer son compte...
 - Avantages : **modularité** naturelle, meilleure lisibilité, **encapsulation**.
- **Classe** : manipuler des objets de même nature. Une classe est un objet et une forme d'espace de nom (*namespace*).
- **Instance** : objet construit sur le modèle d'une classe. Différentes instances d'une même classe disposent des mêmes données et partagent les mêmes méthodes.
- **Modularité** :
 - fonctionnelle : un modèle = fichier source (en C) ou une classe (en C++) ;
 - réutilisabilité : même interface pour des classes aux comportements similaires, solution dans le problème, indépendance entre données et interface.
- **Encapsulation** : indépendance entre programmeur et utilisateur → l'implémentation ou les implémentations (car plusieurs implémentations possibles pour une même interface) et les corrections sont invisibles par l'utilisateur et n'ont aucune influence sur l'interface utilisateur.
Possibilité de rendre *private* un(e) ou plusieurs objets/variables.

```
class Frites
{
    public: // Les méthodes qu'utilise une classe en public

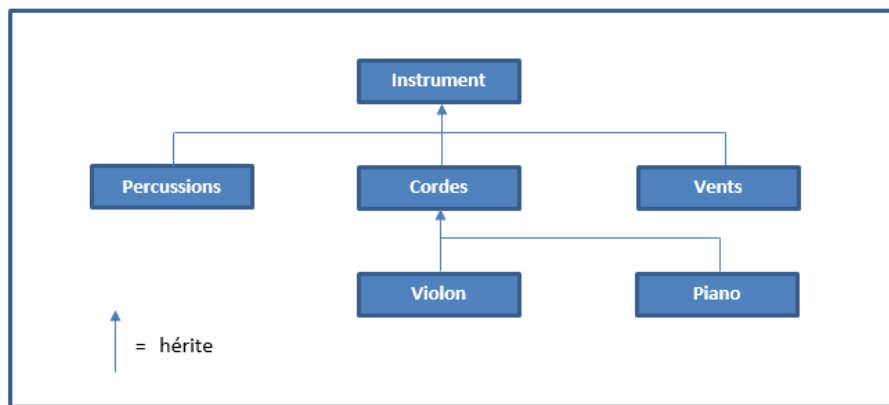
    void eplucher(int nb_patates)
    {

    }

    private: // Tous les attributs à mettre en private

    char type_patate;
    int nb_invites_repas;
    int poids_total_patates;
};
```

- **Polymorphisme** : possibilité pour différentes classes de partager une interface commune et simplification de l'interface (avec réutilisation des conventions).
 - Pour deux classes : possibilité d'avoir des méthodes de même nom.
 - Pour une même classe : possibilité d'avoir plusieurs méthodes de même nom avec des paramètres différents : **surcharge !**
 - ATTENTION : en C, deux fonctions différentes = deux noms distincts. En C++, possibilité d'une seule fonction (par ex.) *echange* pour plusieurs utilisations différentes grâce à la **surcharge**.
- **Héritage** : propriété qui permet de construire de nouvelles classes d'objet à partir de classes existantes. Classe de base > classe dérivée (hérite des attributs et méthodes de la classe de base).
 - À quoi ça sert ? **Enrichir la classe de base.**
 - Utilisations ? Réutilisation de code ; création d'un « protocole » permettant d'uniformiser les classes dérivées ; apport de légères modifications (UI, par exemple) ; test de nouvelles implémentations.



(Voir le cours d'UML pour plus de précisions sur la modélisation concrète de l'héritage.)

(Voir les slides du cours pour des exemples de code relatifs aux concepts exposés dans ce mémo.)

Concepts à maîtriser **absolument** : surcharge (de fonctions, d'opérateurs...) ainsi que les constructeurs.

- **Surcharge** : possibilité d'appeler deux fonctions ayant le même nom, mais effectuant des tâches différentes. **Pas possible en C.** En C, « compterLivres(...) » appelle une seule fonction qui effectue une seule tâche. En C++, on peut appeler deux fonctions différentes (mais portant le même nom) grâce à leurs paramètres. `placerPoints(int x1, int x2)` n'effectuera pas la même chose que `placerPoints(double x, double y)`.
- **Paramètres par défaut** : void donnerTaille(int taille_hauteur = 5, int taille_largeur = 8, int taille_profondeur = 4). Les paramètres par défaut sont 5, 8 et 4. On peut appeler la fonction de la façon suivante : `donnerTaille(12, 7)`. Ainsi, on obtiendra `taille_hauteur = 12`, `taille_largeur = 7` et `taille_profondeur = 4` (ayant conservé sa valeur par défaut vu que l'argument n'est pas donné lors de l'appel à la fonction).

- **Constructeur** : quelques règles :
 - porte le même nom que la classe ;
 - n'a aucun type de retour (« MonConstructeur(); ») : ceci est d'ailleurs un **constructeur par défaut** ;
 - est appelé seulement lors de l'initialisation d'objets ;
 - peut être par défaut ou de copie.

Constructeur de copie = sorte de constructeur surchargé. Permet de dupliquer un objet, entre autres.

```
class MaClasse
{
public:
    MaClasse( const MaClasse & ); /* Le constructeur de copie. Présence
    du const et de la référence, donc bon programmeur C++. */
};
```

Destructeur = nom du constructeur avec un tilde devant → ~MonConstructeur();

Lui non plus ne possède **pas d'argument ni de type de retour**. Il est seulement là pour désallouer en mémoire les objets qui ne sont plus utilisés → on évite les fuites de mémoire, le programme est donc mieux optimisé. Pour les connaisseurs, équivalent non automatique du ramasse-miettes de Java (*garbage collector*).