

Proposition de guide de style pour codage professionnel en C++

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."

--- Martin Fowler, Refactoring: Improving the Design of Existing Code

Les classes : Dans la majorité des cas chaque classe propose 2 fichiers. Un fichier d'entete .hpp qui décrit la classe et un fichier d'implémentation qui code les principales méthodes. Certaines méthodes (inline), les constructeurs et destructeurs peuvent être codés dans le fichier d'entête. Il est recommandé de débiter le nom d'une classe par une majuscule, de même, les attributs et les méthodes **de classes**.

Pour les instances : Pour les attributs et les méthodes d'instance, il est recommandé de débiter leurs noms par des minuscules. Les identificateurs des attributs (ou variables) d'instances peuvent se terminer par un '_'.

Les 'accesseurs' : Les méthodes d'accès / modification peuvent commencer par get/set suivies du nom de l'attribut.

Exemples :

```
Voiture uneVoiture;          class MaClasse
                               {
                               int    _attrInstance;

                               public:
                               int     getAttrInstance() const { return _attrInstance };
                               void     setAttrInstance(const int & inAttrInstance);

                               static float VariableDeClasse ;

ptrSurAvion = new Avion(...);
ptrSurAvion->decole();      };
```

Les méthodes : Chaque méthode peut être précédée d'un bloc de commentaires précisant le nom de la méthode, son rôle, les paramètres en entrée et en sortie ainsi que sa valeur de retour. On peut suivre le guide proposé par un outil de documentation.

```
// ----- //
// nomDeLaMethode      But de la methode      //
// ----- //
// En entrée: les parametres inNomDuParametre1,... //
// ----- //
// En sortie: Explication sur la valeur retournée //
// ----- //
```

Les paramètres de méthodes: Les noms des identificateurs des paramètres en entrée d'une méthode peuvent être précédés d'un préfixe tel que "in". Cette notation se révèle très utile pour le choix des identificateurs similaires notamment quand on manipule des constructeurs.

Exemple:

```
MaClasse::MaClasse (int inX, int inY)      MaClasse * ptrSurUnObjet;
{
    _x = inX;
    _y = inY;
}
int      x = 10;
int      y = 15;

ptrSurUnObjet = new MaClasse(x,y);
```

Les commentaires : Les commentaires monolignes seront préférés et systématiquement employés. Cela permet d'utiliser les commentaires /* */ du langage C pour mettre des portions entières de code en commentaire dans une phase de débogage.

Déclarations de variables locales aux méthodes : Les déclarations sont alignées suivant leurs types. Les pointeurs sont décalés d'un caractère. Par exemple :

```
int      i,j;
Adresse  adresseDeBenny;
char     * ptr;
float     x = 10.45,
          y = 20;
```

Aération (Spacing) : Les codes sources et les instructions sont aérés autant que possible et ce afin de respecter et de mettre en valeur la structure du programme. De même, les commentaires sont alignés autant que possible, aérés et ne doivent pas surcharger inutilement le code. Il convient aussi de sauter des lignes entre chaque bloc « logique » d'un code.

Indentation : Comme en C, les sources doivent être indentés de manière cohérente et rigoureuse. Une indentation avec 3 caractères espace est à la fois lisible et portable (indépendante de la taille des tabulations).

if (condition)	while(condition)	for(condition)	do
{	{	{	{
Bloc alors	Bloc Tant que	Bloc pour	Bloc répéter
}	}	}	} while(condition);
else			
{			
Bloc sinon			
}			

Constantes : En C++ il faut éviter les #define, et préférer les déclarations `const` typées, par contre comme en C, vous pouvez utiliser des majuscules et les '_' pour séparer les noms et faciliter la lecture (Ex : MAX_CHAR).

Largeur du code: pour optimiser la lisibilité du code, il est préférable qu'une ligne de code ne dépasse pas 80 caractères. La vision humaine pour la lecture est plus confortable sur des textes de petites largeurs, la largeur idéale constatée étant moyenne de 8 cm. Une largeur de 80 caractère est déjà bien au dessus de ces 8 cm, plus on code en largeur plus on est contreproductif en ce sens qu'il devient plus difficile d'avoir une vision rapide et globale des lignes de codes que l'on débogue.

Proposition de règles élémentaires de codage C++ et objets :

- Lorsqu'une classe A veut communiquer avec un classe B (une instance de A possède un référence sur B et veut appeler une des méthodes de son Interface) :**

Le fichier entête de la classe A doit utiliser une déclaration anticipative de la classe B : `class B ;`

Eviter absolument toute inclusion du fichier entête de la classe « b.hpp » sauf pour les agrégation par valeur (un objet A comportant directement un autre objet B, et non un pointeur sur un B).

- Le programme principal et toute implémentation d'une classe doit inclure les fichier « X.hpp » de toutes les classes X des objets réellement utilisés par l'implémentation. Tout autre inclusion est superflue et ralentit la compilation**

- Comme en C, tout fichier entête de classe est « gardé » par les instructions du préprocesseur C : #ifndef – De plus il est recommandé d'inclure un bloc de commentaires succinct sur le rôle de la classe**

```
// ----- //
// X.hpp      Classe X : XXXXXX //
// ----- //

#ifndef X_HPP
#define X_HPP

class X
{
    ...
};

#endif
```

Le but de cette « garde » est bien sûr d'éviter les inclusions multiples et imbriquées des fichiers entête.

- Toute classe X qui hérite d'une classe SuperX doit inclure le fichier entête de SuperX dans son propre fichier d'entête, mais pas dans son fichier d'implémentation.**

```
#ifndef X_HPP
#define X_HPP

#include "superX.hpp"

class X : public SuperX
{
    ...
};

#endif
```

```
X.cpp

#include "X.hpp" // SuperX.hpp est
                // donc déjà inclus

X::methode(...)
{
    ...
}
```

- Les codes des templates (déclaration ET implémentation paramétrée) sont forcément donnés dans un fichier entête (.h .H .hxx .hpp) qui doit être inclus et disponible pour compiler le fichier qui utilise un tel template. (Valable pour les templates imbriqués).**