Normes de programmation

Buts des normes :

- Définir des règles permettant d'écrire des programmes qui sont :
  - Corrects.
  - Avec un style consistant.
  - Faciles à lire et à comprendre.
  - Exempts des types d'erreurs communes.
  - Faciles à maintenir par différents programmeurs.
- Le langage C++ est un langage difficile dans lequel la frontière entre une subtilité et un "bug" est très mince.
- Le programmeur a une très grande responsabilité et il doit se discipliner;
- Comme avec le C, le C++ permet d'écrire du code compact mais illisible.
- Le programmeur doit donc suivre les règles décrites dans ce document dans le but d'éviter ces embûches.

Quelques règles :

- **[R1]** Chaque fois qu'une règle est violée, la raison doit être clairement documentée.
- **[R2]** En C++, les fichiers d'interface doivent avoir l'extension **.h**
- **[R3]** En C++, les fichiers d'implémentation ont l'extension **.cpp**.
- **[R4]** Un fichier d'interface ne devrait contenir que la définition d'une seule classe.
- **[R5]** Tous les fichiers doivent avoir un en-tête décrivant le contenu du fichier (voir la section sur les entêtes).
- **[R6]** Tous les fichiers d'interface doivent avoir un mécanisme pour prévenir l'inclusion multiple.

```
#ifndef MACLASSE_H
#define MACLASSE_H
   . . .
  // interface
#endif
```

- **[R7]** Ne **jamais** spécifier le path au complet dans les directives `#include`.

```
// NON!
#include "C:/dev/include/maclasse.h"
// OK!
#include "maclasse.h"
```


- **[R8]** Les sections privées, protégées et publiques doivent être unique et déclarées dans cet ordre.

```
Class CMaClasse
{
private:
   // --- Attributs privés, puis méthodes privées.
protected:
   // --- Attributs protégés, puis méthodes protégées.
public:
   // --- Attributs publics, puis méthodes publiques.
};
```

- **[R9]** Il ne devrait y avoir qu'une seule section publique, protégée et privée.

```
// Mauvais
Class CMaClasse
{
public:
   // --- Quelques méthodes publiques.
protected:
   // --- Attributs protégés, puis méthodes protégées.
public:
   // --- D'autres méthodes publiques. NON!
};
```

- **[R10]** Ne jamais définir des données membres (attributs) dans la section publique de la classe. Les déclarer dans la section privée et définir des méthodes d'accès.

```
// Mauvais
Class CDate
{
public:
     int m_nannee;  // NON!

};
```

```
// Mieux
Class CDate
{
     int getAnnee() const;
public:
     int m_nannee;
};
```

- **[R11]** Les données membres (attributs) d'une classe doivent **toujours** être précédées des caractères **m_** , puis d'un préfixe constitué de **1 à 3 caractères** permettant de préciser leur type.

```
Class CMaClasse
{
public:
     double m_dblvitesse;
protected:
     bool m_blnvide;
     char * m_pszmessage;
};
```

[R12] Les règles de nommage "***Hungarian notation***" des données membres et des variables à appliquer :

Type	Préfixe	Exemple
bool	bln	blnTrouve
byte	byt	bytOctetLu
char	c	cMotif
char *	psz	pszMessage
CPoint	pt	ptOrigine
CRect	rc	rcVue
CString	str	strResultat
CTime(date)	dtm	dtmHeure
CWnd	wnd	wndSaisie
double	dbl	dblTolerance
DWORD	dw	dwNumero
float	flt	fltVitesse
HANDLE	h	hEvent
int	n	nNombre
long	ln	lnPopulation
long double	ldbl	ldblPrecision
LPCTSTR	lpsz	lpszNom
short	sn	snPixel
unsigned char	uc	ucOctet
unsigned int	un	unNbTotal
VARIANT(COleVariant)	vnt	vntCheckSum
WORD	w	wQuantite

- **[R13]** Une méthode qui n'affecte pas l'état d'une classe doit être déclarée **const**. Cette règle doit être **strictement** respectée.

```
Class CMaClasse
{
public:
   int reqDimension() const;
   bool estVide() const;
   bool compareegalite(const CMaClasse & obj) const;
};
```

- **[R14]** Pour le passage de paramètre de type "*Classe*", utilisez le passage de paramètre par référence **constante**.

```
// Mauvais
Class CMaClasse
{
public:
   void nomMethode(CMaClasse uneClasse); // NON!
};
// Bon
Class CMaClasse
{
public:
   void nomMethode(CMaClasse & uneClasse);
};
```


- **[R15]** Une classe alloue et gère dynamiquement des ressources doit :
  - Définir un constructeur correct.
  - Définir un destructeur.
  - Définir un constructeur par recopie.
  - Définir un opérateur d'assignation.

- **[R16]** A l'intérieur d'un opérateur d'assignation, on doit d'abord vérifier l'auto-assignation avant de détruire les ressources.

- **[R17]** Une classe de base qui possède des méthodes virtuelles doit définir un destructeur virtuel.

- **[R18]** Les constantes doivent toujours être définies avec **const** ou **enum**; ne jamais utiliser **#define**.

```
// Mauvais
#define PI 3.14159
// Bon
const double PI=3.14159;
// Mauvais
#define BLEU 0
#define BLANC 1
#define ROUGE 2
// Bon
enum Couleur {BLEU, BLANC, ROUGE};
```

- **[R19]** Ne jamais utiliser de nombres "magiques" dans le code.

```
// Mauvais
if (dblNiveau >= 3.567)
{
   . . .
}
// Bon
const double NIVEAU = 3.567;
// Mauvais
if (dblNiveau >= NIVEAU)
{
   . . .
}
```

- **[R20]** Définir une seule variable par énoncé de déclaration.

```
// Mauvais
char * pstrP, pstrMess;
// Bon
char * pstrP;
char * pstrMess;
```

- **[R21]** Une variable doit être initialisée avant d'être utilisée et si possible, préférer l'initialisation à l'assignation.

- **[R22]** Pour l'allocation dynamique de la mémoire, utiliser les opérateurs **new** et **delete** et non **malloc** et **free**.

```
// Mauvais
int i;
...   // un paquet de lignes de code.
i = 10;
// Bon
int i = 10;
```

- **[R23]** Le nom de la classe possède un préfixe composé d'une lettre majuscule **C**, suivie par le nom de la classe qui doit commencer par une lettre majuscule. Si le nom de la classe est un mot composé, alors la première lettre des mots est aussi en majuscule.

```
// Mauvais
class donneeutile;
// Bon
class CMaClasse;
class CSectioCritique;
```

- **[R24]** Un nom de méthode ou de fonction doit être de la forme <verbe + complément> et exprimer clairement ce que fait la méthode ou la fonction.

- **[R25]** Les noms de constantes sont toujours définies en lettres majuscules. Les mots composant le nom doivent être séparés par des soulignés _.

```
const int VALIDE = 1;
enum Couleur {BLEU, BLANC, ROUGE};
const double NIVEAU_CRITIQUE = 3.567;
```

- **[R26]** Les variables utilisées comme compteurs de boucle sont les lettres i, j, k, l, m, ... et elles sont définies au moment de leur utilisation uniquement.

```
for (int i=0; i<10; i++)
{
     for (int j=0; j<20; j++}
     {
          . . . // --- code à exécuter
     }
}
```


- **[R27] L' indentation** permet de visualiser la structure d'un programme. Une indentation trop faible nuit à la lisibilité, tandis qu'une indentation trop grande décale de façon inutile tout le code vers la droite. La norme sera donc une indentation de **trois espaces**.

- **[R28] L'imbrication** poursuit le même but que l'indentation, soit d'augmenter la lisibilité du code. Laisser un espace après les énoncés de contrôle.

```
for (int i=0; i<10; i++)
{
     for (int j=0; j<20; j++}
     {
          . . . // --- code à exécuter
     }
}
```

- **[R29]** Le code pour le traitement normal devrait se retrouver sous le **"if"**. Le code pour le traitement des erreurs ou exceptions se retrouve sous le **"else"**.

```
if (!nErreur)
{
  // --- traitement normal
}
else
{
  // --- traitement d'exception
}
```

- **[R30]** Le commentaire global **du fichier d'interface .h** devrait contenir :
  - Le nom du fichier.
  - Le nom de la classe.
  - La description de la fonctionnalité de la classe.
  - La liste des attributs de la classe avec description.
  - Une note indiquant les particularités d'implantation s'il y a lieu.
  - Date de création et nom de l'auteur.
  - Dates de modification, description et nom du programmeur.

```
//*****************************************************************
// Fichier         : ifchain.h
//
// Classe          : Cifchaine
// Description     : Gestion simple des chaînes de caractères.
// Attributs       : char * m_pszchaineP : Chaîne terminée par '\0'.
//
//
// Notes           : Cette implémentation n'est pas optimisée.
//
//*****************************************************************
// 01-09-03        Ph. Better  Version Initiale.
// 07-09-03        Ph. Better  Ajout de ceci et de cela.
//*****************************************************************
```

- **[R31]** L' entête du fichier d'implémentation .cpp doit contenir :
  - Le nom du fichier.
  - Le nom de la classe.
  - Date de création et nom de l'auteur.
  - Dates de modification, description et nom du programmeur.

```
//*****************************************************************
// Fichier        : ifchain.cpp
//
// Classe         : Cifchaine
//*****************************************************************
// 01-09-03       Ph. Better  Version Initiale.
// 07-09-03       Ph. Better  Ajout de ceci et de cela.
//*****************************************************************
```

- **[R32]** Chaque méthode implémentée doit être accompagnée d'un entête constitué par :
  - Une description de la méthode
  - Paramètres d'entrée avec description.
  - Paramètres de sortie avec description.
  - Note pour le programmeur s'il y a lieu.

```
//*****************************************************************
// Description    : cette méthode fait ceci et cela.
//
// Entrée         : Type nom :      description
//
// Sortie         : Type nom :      description
//
// Note           :
//*****************************************************************
```