TANOOKI
Le coaching pour tous

NORMES DE DEVELOPPEMENT

Langage C++

PROPOSITION COMMERCIALE

Client – Formation Supérieure

N° - TNK/DEV/020190601
Date – Janvier 2019

VOTRE CONTACT

Cédric Obejero
+33 (0)6 78 68 71 29
cedric.obejero@tanooki.fr
www.tanooki.fr

Clause de confidentialité

Toute information contenue dans le présent document strictement confidentiel est fournie dans le seul but de répondre aux enjeux de formation, et ne peut être utilisée à d'autre fin.
Tout destinataire s'engage à ne diffuser ni faire connaître tout ou partie des informations contenues dans le présent document à quelque tierce partie que ce soit, sans l'autorisation écrite préalable de la société TANOOKI.

SOMMAIRE

1. Présentation

Ce cahier qualité décrit les normes générales de programmation en langage C++ applicables sur le projet.
Ces normes concernent la forme et le fond :

- la forme, pour la présentation des sources, les règles de nommage,
- le fond, pour la structuration du code, les instructions à éviter, le style recommandé.

Le présent document constitue un point de départ, susceptible d'être complété ultérieurement.

2. Formatage du code

2.1 Les fichiers

2.1.1 Nommage

- Le nom d'un fichier doit toujours être écrit en minuscules, pour éviter les problèmes de distinction de casse sur certains systèmes.
- Le nom des fichiers source sera préfixé par le nom de la classe à laquelle il se réfère, si possible, et suffixé par ".cpp" (en minuscules).
- Le nom des fichiers d'en-tête sera préfixé par le nom de la classe à laquelle ils réfèrent ou par le nom du module, et suffixé par ".h" (en minuscules). Les fichiers d'en-tête
- La règle du nommage 8.3 n'est plus à l'ordre du jour, car abandonnée sur la quasi-totalité des systèmes actuels.
- N'utiliser que les caractère alphabétiques (a-z), numériques (0-9) ou le caractère de soulignement (_). N'utiliser que des caractères alphabétiques pour le premier caractère.
- Eviter les abréviations ou les acronymes obscurs et trouver un nom simple et explicite, dont la compréhension est intuitive.

2.1.2 Nombres et volumes

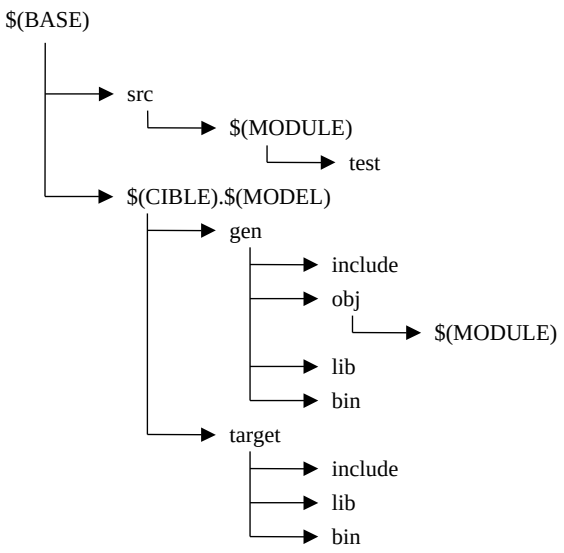
On se contentera de fixer les bornes raisonnables suivantes :

- Un logiciel bien structuré regroupe en librairies ses classes selon des règles d'homogénéité technique ou fonctionnelle. Une même librairie ne devra pas exporter plus de 20 classes hors exceptions.
- La règle vaut pour un exécutable fabriqué directement à partir de modules objets sans librairies.
- Un fichier source ne devra pas excéder 100 Ko ou 20 pages de listing, ou encore 2000 lignes.

2.1.3 Emplacements (environnement)

Noms des répertoires, pour le développement ET l'arborescence de fonctionnement : mêmes règles que pour les fichiers : minuscules obligatoires, noms simples et significatifs.

L'arborescence de développement standard est celle mise en place sur le serveur d'archivage de référence du projet.



- $(XX) désigne une entrée duplicable.
- Les fichiers .cpp et .h sont dans le même répertoire src/$(MODULE).
- Les autres répertoires include (sous gen et sous target) ne concernent que les .h exportés par les modules.
- Tous les objets, d'où qu'ils proviennent, doivent aller sous obj.
- Chaque module contient un répertoire test contenant les sources d'un ou plusieurs outils de test du module. Ces outils servent à tester le module de façon unitaire.
- Les fichiers générés par les makefile (copies de .h, fichiers objets) sont copiés dans le repertoire $(CIBLE).$(MODEL). La cible peut être win32, sol26, aix43 … (nom du système suivi de son numéro de version). Le modèle est généralement wrk (avec trace et librairie de gestion des allocations mémoires) ou rel (sans outils de débogage), mais peut avoir d'autres valeurs (insure par exemple).
- Le répertoire $(CIBLE).$(MODEL) contient deux répertoires target et gen. target contient ce qui sera livré (binaires, .h livrables …) et gen contient les fichiers qui ne seront pas livrés (fichiers objets, .h non-livrables …).

2.1.4 Les cartouches

2.1.4.1 Cartouche de fichier

```
/************************************************************/
 *
 * Copyright (c) <CLIENT> - contrat n° XXXXXXXXX
 * Reproduction et diffusion interdites.
 * Developpe par TANOOKI SASU
 * Projet <nom>
 *
 * SOUS-SYSTEME : <nom du module>
 *
 * SOURCE       : <nom du fichier .h ou .cpp>
 * PRESENTATION : <description du fichier en une ligne>
 *
 * AUTEUR       : <trigramme de l'auteur du fichier>
 * VERSION CVS  : $REVISION$
 * DATE         : <date de creation>
 *
 ************************************************************/
```

Les OBSERVATIONS devront impérativement contenir une explication des cas de compilation conditionnelle (introduits par #ifdef) si applicable.
Important : les fichiers source .cpp devront en outre contenir la ligne suivante, utilisée par CVS (système de gestion de versions) et par les commandesde type what :

```
static const char * __cvs_id="@(#) $Header$" ;
```

2.1.4.2 Cartouche de classe

Dans ce cartouche, on ne décrit que les méthodes publiques.

```
/*************************************************************
 * CLASSE       : <nom de la classe>
 * PRESENTATION : <description de la classe et de sa fonction>
 *
 * METHODES PUBLIQUES :
 * <Methode1>   : <description rapide de la methode 1>
 * <Methode2>   : <description rapide de la methode 2>
 * <Methode3>   : <description rapide de la methode 3>
 * <Methode4>   : <description rapide de la methode 4>
 *
 * OBSERVATIONS :
 *************************************************************/
```

2.1.4.3 Cartouche de méthode

```
/*************************************************************
 * METHODE      : <nom de la classe>::<nom de la methode>
 * PRESENTATION : <description de la methode et de sa foncion>
 *
 * ENTREES : <parametre_1> : <presentation du parametre 1>
 *           <parametre_2> : <presentation du parametre 2>
 *           <parametre_3> : <presentation du parametre 3>
 *
 * SORTIES : <parametre_4> : <presentation du parametre 4>
 *           <parametre_5> : <presentation du parametre 5>
 *           <parametre_6> : <presentation du parametre 6>
 *
 * RETOUR  : <valeur_1> : <signification>
 *           <valeur_2> : <signification>
 *           <valeur_3> : <signification>
 *
 * EXCEPTIONS : <exception_1> : cas d'erreur 1
 *              <exception_2> : cas d'erreur 2
 *              <exception_3> : cas d'erreur 3
 *
 *************************************************************/
```

Pour les fonctions, on utilisera le même cartouche que pour une méthode en remplaçant "METHODE" par "FONCTION".

Si le fichier requiert des directives de compilation ou d'édition de liens particulières, il est impératif de les citer dans le cartouche. On explicitera notamment tous les cas de compilations conditionnelles (introduits par #ifdef).

La description d'une méthode doit expliquer ce que fait la méthode, et non entrer dans des détails techniques sur la manière dont elle le fait. Des remarques de cet ordre ont leur place dans la rubrique « Observations ».

2.1.5 Format général des fichiers

2.1.5.1 Disposition des Données dans les Classes

Que ce soit dans les déclarations ou les implémentations, l'ordre des données dans les classes sera le suivant :

1. attributs privés,
2. attributs protégés (protected),
3. attributs publics,
4. méthodes privées,
5. méthodes protégées,
6. méthodes publiques.

2.1.5.2 Déclarations

L'ordre des déclarations est le suivant :

- les #include, dans cet ordre :
  - les .h système,
  - les .h applicatifs généraux,
  - les .h applicatifs spécifiques ;
- les #define de valeurs qui peuvent être utilisés dans des déclarations de variables (tailles) ;
- les variables statiques ;
- les fonctions inline (suivies de leur implémentation, qui doit se trouver dans le fixhier d'en-tête) ;
- enfin, les macros (à délaisser au profit des fonctions inline, cf. 3.1).

2.1.5.3 Méthodes

L'ordre des méthodes est important, et il faut toujours partir du principe que le compilateur est à 'une passe' (même s'il en fait parfois plusieurs, il est plus facile pour le lecteur de travailler en une seule passe).
On commencera par le constructeur, puis par le destructeur quand ceux-ci existent. On respectera un ordre logique, s'il y en a un, et on regroupera au maximum les méthodes ayant un rapport entre elles. L'ordre des méthodes dans le .h et dans le .cpp devra être le même, et les méthodes privées devront être séparées des méthodes publiques.

2.2 Règles d'écriture

Aucun accent n'est toléré dans un fichier source. Cette règle ne connaît qu'une et une seule exception : la mention légale du cartouche de fichier contenant le mot « Télécom ».

2.2.1 Nommage des Constantes

Tous les noms des constantes sont écrits en majuscules.

On peut considérer deux types de constantes en C++ :

- Les constantes déclarées par l'intermédiaire d'un #define. Ces constantes à la mode C doivent être utilisées quand elles sont très générales (par exemple : les tailles fixes communes à tout le module). Elles sont écrites en majuscules et préfixées par le nom du module (le séparateur est le caractère de soulignement) : MODULE_LG_HEADER, …
- Les constantes déclarées par l'intermédiaire d'un const static à l'intérieur d'une classe. Ces constantes ont l'avantage d'être typées, et spécifiques à une classe. Elle ne sont pas préfixées. Elles seront utilisées de la façon suivante : RC_OK à l'intérieur de la classe, CMaClasse::RC_OK s'il s'agit d'une constante publique utilisée à l'extérieur de la classe.

On respectera par ailleurs les préconisations suivantes :

- S'il existe plusieurs constantes représentant les différentes valeurs possibles pour une donnée (paramètre d'une fonction, critère de sélection d'un « switch », ...), alors on préfixera tous les noms avec un mnémonique faisant référence à l'objet.
- Ce mnémonique sera séparé du nom de la constante par un 'underscore' '_', et le nom sera entièrement en majuscules.

Par exemple, un ensemble de constantes représentant les différents types de fichiers pourrait regrouper les noms suivants : TF_BINAIRE, TF_SEQTEXTE, TF_DBASE, etc.
Si une constante est isolée, son nom sera écrit entièrement en majuscules, sans préfixe et sans caractère de séparation.
Dans certains cas, pour améliorer la lisibilité de la constante, certains mots pourront être séparés par un délimiteur, qui sera alors un underscore. Cet usage est toutefois à limiter aux noms assez longs.
Exemples : inutile pour CT_VERT_CLAIR (CT_VERTCLAIR est lisible), mais utile pour TAILLE_MAXIMALE_FICHIER (TAILLEMAXIMALEFICHIER est moins lisible).

2.2.2 Nommage des Variables et Arguments

Tout nom de variable doit être préfixé par une minuscule indiquant son type :

PREFIXE	TYPE
p	Pointeur
r	Référence
t	Tableau
u	Unsigned
b	Byte
c	Char
f	Flag (booléen)
i	Integer
l	Long
ll	Long long
d	Double
s	Short
w	WORD
dw	DWORD
st	Size_t
sz	Chaîne se terminant par '\0'

Cette règle devra notamment être impérativement respectée pour les variables globales externes et privées, les arguments de fonctions ou méthodes, et pour tous les objets exportés.

Les noms de membres de structures et des attributs de classes respecteront les mêmes conventions.

Les noms de variables de type structure ou instance de classe ne sont pas préfixés.

Les noms des arguments d'une fonction ou méthode seront écrits en minuscules, la première lettre de chaque mot en majuscules, sans séparateur.

On pourra réserver l'usage de certains noms à des emplois particuliers : i, j et k sont admis pour des indices de boucles ou des iterators, rc pour des codes retours, etc., mais uniquement dans des cas généralement admis dans le monde des développeurs C++.

La différence entre une variable préfixée par t ou par p peut être subtile. dans le cas suivant :

```
void MaFonction(char pMonTableau[10]);
```

Il s'agit d'un pointeur plutôt que d'un tableau (lors du passage du paramètre, ce qui est transmis est le pointeur et non le tableau entier). La règle généralement admise et que sizeof(pMaVariable) doit renvoyer 4 (le nombre d'octets d'un pointeur pour un système 32 bits) alors que sizeof(tMaVariable) doit renvoyer la taille totale du tableau ou de l'instance en octets.

Les noms des variables ou attributs statiques seront écrits en minuscules, la première lettre de chaque mot en majuscules, et sans séparateur underscore ou autre. Les variables statiques devront être préfixées par le nom du module, tandis que les attributs statiques devront être préfixés par la lettre s minuscule et un caractère de soulignement (s_) seulement.

Les noms des variables ou attributs publics suivent les mêmes règles que les variables statiques en remplaçant le préfixe par la lettre p minuscule et un caractère de soulignement (p_).

```
Exemples :     ModuleNomGare, ModuleTailleMessage
               s_NomGare, s_TailleMessage, p_TailleMessage
               sp_TailleMessage (statique et public)
```

Les attributs non-statiques seront écrits de la même façon que les attributs statiques,sans la lettre s.

```
Exemples :_NomGare, _TailleMessage
```

Les noms des variables locales (variables déclarées dans une méthode) seront écrits en minuscules, sans majuscules pour la première lettre, et avec le séparateur underscore.

```
Exemples : nom_gare, taille_message
```

Si une abréviation est utilisée, elle doit l'être systématiquement et rester la même partout (par exemple, si « numéro » est abrégé en « No », on ne doit jamais voir « Num » ou « Numéro », ou encore « Number »).
Voici un exemple qui illustre l'intérêt du préfixage des noms de variables, et qui permet de garder une même appellation générale pour un même objet :

```
/* Variables globales */
FILE     FichierCourant;
FILE     *pFichierCourant;
/* Variables locales */
FILE     fichier_courant;
FILE     *p_fichier_courant;
```

Toutes les attributs et variables globales doivent être *commentées*. Tout commentaire, même court ou évident, vaut mieux qu'aucun commentaire. Le commentaire ne doit pas tant expliquer le contenu de la variable que la *manière* dont elle est utilisée.

2.2.3 Nommage des Méthodes et Fonctions

Les noms de méthodes ou de fonctions suivent les mêmes règles que les noms de variables, mais elles ne sont pas préfixées par types. Ces noms doivent désigner une action, soit à l'aide d'un substantif clair (« Initialisation() »), soit à l'aide de verbes (« OuvrirFichier() »). Par contre, les fonctions sont préfixées par le nom du module tandis que les méthodes ne le sont pas (de manière générale, il n'est pas utile de préfixer tout ce qui est contenu à l'intérieur d'une classe, puisque vu de l'extérieur, cet attribut ou cette méthode seront vus par l'intermédiaire de l'opérateur :: de la façon suivante : CModMaClasse::_MonAttribut).

Les méthodes ou les fonctions ne sont pas préfixées par leur type, cependant les méthodes statiques seront préfixées par la lettre s minuscule suivie d'un caractère de soulignement (s_).
Les méthodes ou fonctions inline ne doivent pas contenir de signe distinctif, ce qui permet de rajouter ou d'enlever le caractère inline dans une phase d'optimisation sans plus de changement. Cependant, il est possible dans de rares cas de les écrire en majuscules, de la même façon qu'une macro, puisqu'elles s'en rapprochent fortement.

```
Par exemple : MaMethode, s_MaMethode.
```

2.2.4 Nommage des Classes

Les classes sont toujours préfixées par la lettre E pour les classes d'exception, I pour les interfaces, C pour les autres classes et par le nom du Module, et leur nom écrit en minuscules et initiales en majuscules.

```
Par exemple, pour un module MOD : CModMaClasse
```

2.2.5 Les Commentaires

En dehors des utilisations déjà indiquées, si les règles d'écriture sont respectées, peu de commentaires sont réellement utiles dans un programme. En effet, la lisibilité même des noms rend inutiles des commentaires du type de cet exemple :

```
fo(ft);  // Ouverture du fichier de travail
OuvrirFichier(szNomFichierTravail);    // Meilleur!
```

Par contre, les commentaires sont toujours les bienvenus pour attirer l'attention sur des points particuliers où la lecture seule du code source, aussi soigné soit-il, ne suffit pas à comprendre ce que fait le programme.

Il est recommandé de rappeler le nom d'une méthode, d'une classe ou d'une fonction en regard de son accolade de fin, ce qui facilite les recherches automatiques.

```
void mafonction(void)
{
...
} /* mafonction */
```

De même pour pour une boucle ou une imbrication de plus d'une vingtaine de lignes, ou des #endif, fins de switch, while, etc.

On rappelle que les caractères accentués sont à proscrire dans les commentaires, comme partout ailleurs.

2.2.6 Langue

On évitera autant que possible le « franglais » et tous les mélanges de langues, pour un même nom. A proscrire : FICHIER_SIZE_MIN, et prendre MIN_FILESIZE ou TAILLE_FICHIER_MIN.

Pour tous les objets exportés d'un module, constantes, variables, fonctions, etc... on respectera l'uniformité de langue : si le choix est l'anglais, alors tous ces objets (classes, méthodes …) doivent avoir une appellation en anglais. Idem pour le français. A l'intérieur d'une librairie, et pour des objets non exportés, cette règle n'est pas impérative.

Pour tous les commentaires et les cartouches, la langue imposée est le français.

2.2.7 Présentation du code

Il faut de l'aération... sans toutefois tomber dans l'excès.

Quelques règles simples :

- Une fonction ou une méthode ne doit pas excéder 2 pages de listing. Même complexe, on peut toujours la découper ;
- Un saut de ligne doit être mis avant et après chaque cartouche ;
- Un saut de ligne doit être mis entre deux paragraphes importants de code ;
- Dans un prototype de fonction, on sautera une ligne à chaque paramètre ;
- Dans un appel de fonction, la liste des paramètres devra tenir sur une ligne. Si ceux-ci sont trop nombreux, on sautera une ligne à chaque paramètre ;
- Le code source doit être indenté, l'indentation de base étant fixée à 2 caractères. Aucun caractère de tabulation ne doit être utilisé (la majorité des éditeurs permettent de remplacer la tabulation par son

équivalent en espaces), sauf dans les commentaires. On peut toutefois utiliser la tabulation dans le code source quand cela est indispensable. La tabulation sera de 8 espaces ;

- Eviter les indentations imbriquées trop nombreuses, comme des 'if' imbriqués sur 10 niveaux. Une bonne limite est 3 niveaux ;
- On ne définira qu'une variable par ligne ;
- De manière générale, on évitera le code obscur et on favorisera lisibilité et découpage ;
- La largeur d'une ligne est limitée à 132 caractères ;
- Utilisation des accolades :
    - dans le cas d'une définition de fonction, aller à la ligne avant l'accolade de début du corps de la fonction, comme suit :

```
int Fonction(int iParam)
{
...
}
```

    - dans tous les autres cas, comme par exemple après les instructions if ou switch, l'accolade ouvrante est placée en fin de ligne, sans aller à la ligne, comme suit :

```
if (iParam > 0) {
...
}
```

- Quand on utilisera un if, on placera d'abord le cas anormal :

```
   if( ) {
     // on place ici le cas anormal
   } else {
     // et ici le cas normal
   }
```

- On n'utilisera qu'une instruction par ligne. On remplacera par exemple :

```
j = tableau[i++];
if((cr = fonction()) == CR_OK) { …
par :
j = tableau[i];
i ++;
cr = fonction();
if (cr == CR_OK) { …
```

3. Structure du code

3.1 Macro-Instructions & Méthodes inline

Les méthodes ou fonctions inline offrent les mêmes performances qu'une macro en fournissant les avantages d'un appel de fonction : vérification du type des paramètres, passage par valeur ou référence. On utilisera donc au maximum des méthodes ou fonctions inline, tout en conservant les macros habituelles comme la macro de trace TR par exemple.

Lors de la définition d'une macro-instruction, il faut systématiquement utiliser des parenthèses pour séparer les paramètres, ceci afin d'éviter toute ambiguité à la compilation.

Eviter les macros avec plusieurs instructions sur plusieurs lignes.

Rappelons que les méthodes ou fonctions inline doivent être totalement définies à l'intérieur du fichier d'en-tête .h, par exemple de la manière suivante :

```
/* exemple de methode inline */
class CMaClasse
{
  private :
    …
  public :
    inline void LaMethodeInline(void);
    …
};

inline CMaClasse::LaMethodeInline(void)
{
  // ici le code est place apres la declaration de la classe
  …
}
```

3.2 Les types entiers

Il n'existe malheureusement pas de norme concernant l'implémentation des short, int et long. Il est seulement admis qu'un short fait au moins 16 bits, un long au moins 32 bits, et qu'un short est plus petit (ou de même longueur) qu'un int, lui-même plus petit ou de même longueur qu'un long.
Lorsque la taille en nombre de bits est critique (par exemple, pour la cryptographie), et seulement dans ces cas, on utilisera les types définis dans le fichier commun.h : BYTE, UINT16, UINT32, UINT64.

3.3 Les valeurs en « dur »

On ne mettra jamais de chemin d'accès en dur, y compris dans les makefiles (on passera toujours par un élément de configuration, provenant d'un fichier ou d'une variable d'environnement).

On ne mettra pas non plus de valeurs en dur (on utilisera des éléments de configuration ou des constantes), en particulier pour la taille des tableaux.

Toujours utiliser une constante pour définir la taille d'un tableau destiné à contenir une donnée spécifique dont la longueur maximale est connue, que ce soit pour une allocation statique ou dynamique.

Dans le cas de chaines 'sz' (terminées par '\0'), faire mention explicite du '+1' pour le zéro binaire :

```
#define LG_FILENAME  255
char szFileName [LG_FILENAME + 1];
```

On utilisera systématiquement le sizeof() sur la variable plutôt que sur le type (mais attention aux pointeurs !).

3.4 Initialisation des variables

Les variables devront être définies au début de la fonction, avant le code. Elles devront impérativement être initialisées dès qu'elles sont définies, de façon à ne jamais avoir de variables dans un état indéfini.

Pour les constantes de type 'texte', les directives #define ne sont pas toujours la bonne solution pour l'initialisation. En effet, le préprocesseur va remplacer la directive par la valeur de la chaîne à chaque fois qu'il la rencontre. Dès qu'un #define 'texte' est rencontré 2 fois, il y a donc double occupation de place pour une même chaîne invariable. Il vaut mieux alors utiliser une variable globale initialisée avec la valeur de la chaîne constante.

Il est interdit d'allouer des objets sur la pile, afin d'éviter une saturation particulièrement préjudiciable pour des applications multithread. La procédure à employer est l'utilisation d'un autopointeur (classe auto_ptr de la librairie standard). C'est l'autopointeur qui sera pris sur la pile, et prendra en charge les libérations, soit en fin de bloc, soit en cas de sortie sur une exception. Un exemple est donné ci-dessous :

```
void f(CPoint p1, CPoint p2)
{
  auto_ptr<Rectangle> pRec(new Rectangle(p1, p2); // pRec pointe sur un Rectangle

  pRec->Remplit(noir);     // p s'utilise comme un pointeur ordinaire
  if (dans_le_petrin) throw Petrin() ;
  ...
}
```

3.5 Les commutateurs d'instruction

Une clause switch devra toujours contenir un traitement pour le cas default, même si ce cas apparaît a priori impossible et que le traitement doit se limiter à l'émission d'une simple trace, voire à un commentaire.
Un fall-through dans un case devra toujours être explicitement commenté. De façon plus générale, les fall-throughs (pas de break dans un case) sont à éviter :

```
switch ( code ) {
case CODE_1 :
case CODE_2 :
  // traitement du premier cas
  break;

case CODE_3 :
  // traitement du second cas
  // ici on commente pourquoi il faut un fall-through (pas de break)

case CODE_4 :
  // traitement du troisième cas
  break;

default :
  // ce cas ne doit pas survenir
  // émission d'une trace et renvoi d'une erreur
}
```

3.6 Manipulation des variables

Si le type de donnée impose un suffixe à sa valeur (120L pour un 'long'), il ne faut pas l'omettre.

Eviter les affectations dans les tests, et, de façon plus générale, dans les expressions.

```
'if (rc=read...' ou 'mafonction(param=1, ...' sont à proscrire.
```

3.7 Fichiers d'En-Tête

Protéger les fichiers d'en-tête contre les doubles inclusions :

```
#ifndef MON_INCLUDE
#define MON_INCLUDE
/* ... contenu du fichier... */
#endif /* MON_INCLUDE */
```

Si le programme utilisateur doit avoir inclus au préalable d'autres fichiers d'en-tête, lister ces prérequis au tout début, dans le cartouche du fichier.

L'inclusion de fichiers en-tête dans d'autres fichiers en-tête est fortement déconseillée. Cependant, il est toléré d'inclure dans un fichier d'en-tête de module des fichiers d'en-tête strictement internes au module (types et constantes privés, etc.), et dont l'utilisateur du module n'a pas à avoir connaissance.

3.8 Méthodes et Fonctions

Des fonctions trop grandes sont source d'erreurs. Il est rarissime qu'une fonction très longue ne soit pas décomposable en plusieurs fonctions.

Il ne faut mettre qu'un seul point de sortie par méthode ou fonction, ce qui facilite grandement la relecture et surtout la pause de points d'arrêts sous débogueur.

Sur les méthodes ou les fonctions appelées, tester systématiquement le code retour. (si l'absence de test est justifiée on omettra alors celui-ci et on ajoutera (void) devant l'appel en n'oubliant pas un commentaire explicite).

Limiter si possible le nombre de paramètres pour une fonction ou une méthode à 8.
Il est important de toujours avoir conscience que toute méthode, fonction, constructeur ou opérateur est susceptible de lancer une exception, et que toute exception doit être traitée ou faire partie des exceptions listées et détaillées dans le cartouche.

Une méthode ou une fonction ne doit pas renvoyer (par l'intermédiaire du code retour ou d'un paramètre de sortie) de pointeur sur un objet qui a été créé à l'intérieur de celle-ci.

Il est recommandé d'utiliser la méthode de l'invariant, afin de faciliter la détection d'erreurs. L'invariant est le fondement de la preuve de programme, science destinée à garantir le résultat d'un traitement de données. Il indique un ensemble de conditions qui doivent rester vraies tout au long de l'exécution. Une assertion permet de vérifier un invariant de manière simple, dans un code source. Un invariant pour une méthode peut par exemple s'écrire : « Soit un pointeur est nul, soit il pointe vers une zone de mémoire valide (i.e. non libérée) ».

Dans l'idéal, une méthode commence et se termine par un ASSERT() pour :

- s'assurer que la méthode se trouve dans un état valide au moment ou on l'applique,
- s'assurer que la méthode laisse la situation dans un état valide.

Cela ne constitue pas une règle, mais ces techniques peuvent être utile dans des portions de code sensibles.

3.9 Habitudes "C" à Perdre

De façon générale, on laissera tomber les fonctions C au profit des opérateurs ou des classes C++ : new au lieu de malloc, cout plutôt que printf, etc.

3.10 Pratiques "C++" à Proscrire

Eviter de redéfinir des opérateurs. La redéfinition d'opérateurs, si elle peut être parfois élégante, est extrêmement risquée car génératrice d'erreurs et de traitements cachés dont l'utilisateur n'a pas toujours conscience.

Certaines pratiques relevant plus d'un formalisme excessif que d'une utilité réelle sont à proscrire. Citons notamment :

- les classes fournissant des méthodes get et set qui sont seules habilitées à manipuler les attributs de la classe. Cela impose, si on veut renommer un attribut, de renommer les deux méthodes associées, multipliant le nombre de modifications à effectuer par trois.

Cette manière de procéder peut se justifier, mais ne doit pas être systématiquement employée. Dans le cas où l'on désire proposer de telles fonctions, il est préférable de les déclarer inline.

- les méthodes contenant le nom d'un attribut. L'esprit de la programmation orientée objet est de masquer ce qui n'est pas public, de façon à limiter les modifications de l'interface (les méthodes) qui se répercutent sur tout le code qui utilise la classe en question. Les noms des méthodes devront être choisis en fonction de ce qu'attend l'utilisateur de la classe, et pas en fonction de la manière dont la classe a été conçue.

3.11 Les Exceptions

Les exceptions sont conçues pour gérer les cas anormaux. Il s'agit d'erreurs inattendues, qui en principe « ne doivent pas » survenir (par exemple, la mémoire est épuisée, ou une panne de réseau a interrompu une communication). Dès lors qu'un cas d'erreur peut être considéré comme fonctionnel, il sera signalé d'une autre façon (l'absence d'un paramètre dans un fichier de configuration, par exemple, ne donne pas lieu à une exception).

Dans le cadre du projet, on utilisera une classe d'exceptions propriétaire C<PROJECT>Exception, qui dérive de la classe exception fournie par le langage C++. Pour plus de détails, se référer au document de spécifications de C<PROJECT>Exception.

Il faut avoir conscience qu'une exception peut être levée depuis n'importe quelle partie du code. L'écriture du code devra se faire en gardant ce principe à l'esprit. Les seules dérogations à cette règle sont quelques opérateurs ou méthodes dont on peut être certain qu'ils ne lèveront aucune exception :

- pointeur_1 = pointeur_2;
- delete pointeur;
- auto_ptr::release();

3.12 Opérateurs de cast

L'utilisation de const_cast est interdite.

L'utilisation de reinterpret_cast est également proscrite. Tout comme dans le cas de const_cast, son besoin traduit dans la grande majorité des cas un problème de conception.

L'utilisation de static_cast pour remonter le schéma de dérivation (transformer une classe fille en classe mère) et de dynamic_cast pour redescendre le schéma de dérivation (transformer une classe mère en classe fille) est conseillée.
On évitera les conversions d'objets, on n'utilisera que les conversions de pointeurs. Une des justifications de cette règle est le comportement en cas d'erreur : une erreur sur un dynamic_cast de pointeur renverra le pointeur NULL, tandis qu'une erreur sur un dynamic_cast de référence ou d'objet se traduira par une exception bad_cast.

De façon générale, le nombre de casts sera le plus faible possible, et passera toujours par un des deux opérateurs dynamic_cast et static_cast.

3.13 Les Dérivations

Le nombre de niveaux de dérivation est limité à 3.

Le temps que l'on doit passer à coder un module dépend du temps que l'on va passer à l'utiliser. Il ne sert à rien de généraliser de manière excessive, dans l'éventualité d'une réutilisation hypothétique, des classes qui ne seront utilisées qu'une fois.

3.14 Les Templates

On n'utilisera pas de templates en dehors de ceux de la Standard Template Library.

L'utilisation de templates classiques de la STL devra respecter les règles suivantes :

- Les listes, queues et autres conteneurs ne devront contenir que des pointeurs sur objet, et non des objets eux-mêmes. Cette règle vise à éviter les appels effectués par la STL, parfois à l'insu de son utilisateur, à des constructeurs par recopie, destructeurs et autres membres de la classe contenue.
- Les seules opérations autorisées sont :
    - ajout d'élément ;
    - suppression d'élément ;
    - parcours avec itérateur ;
    - effacement de la liste (méthode clear) ;
    - recherche, dans le cas du template map.

3.15 Les Classes abstraites

Les classes d'interface ou classes abstraites permettent d'exprimer les abstractions sous-jacentes d'un système. Elles sont essentielles lorsque l'on manipule des concepts, sans se soucier du type exact du pointeur ou de l'objet qui prendra effectivement la place de ce concept au runtime.

Prenons l'exemple d'un module fournissant une API et d'un utilisateur de cette API, qui se conforment tous deux à une interface. Cela se traduit concrètement de la manière suivante :

Du point de vue de l'API

Le module API exporte le fichier d'en-tête d'une classe abstraite (ou classe d'interface). Dans son code, elle manipule des objets de cette classe de base.

Du point de vue de l'utilisateur

L'utilisateur inclut le fichier d'en-tête de la classe abstraite (ou classe d'interface). Il dérive une classe de cette classe abstraite, en implémente les méthodes virtuelles pures, et manipule dans son code des instances de la classe dérivée.

Sans entrer dans les détails, on peut citer de gros avantages de ce formalisme :

- il permet d'isoler les modules entre eux en se concentrant sur une description aussi fine que possible de leurs interactions, d'où une plus grande modularité et une propagation des modifications restreinte ;

- il permet d'ajouter des fonctionnalités en ajoutant du code, mais sans jamais avoir à éditer du code existant, réduisant considérablement le risque d'introduction d'erreurs.

Un exemple concret peut être celui d'une librairie de gestion d'événements et de timers. On pourra définir deux classes :

- Une classe abstraite IEvent, contenant deux méthodes virtuelles pures servant au call-back : OnTimeOut() et OnEvent() ;
- Une classe CMoteur contenant une méthode AddEvent prenant deux paramètres : un pointeur sur un IEvent et la durée de validité de cet événement.

La classe Cmoteur remplit deux fonctions: appeler la méthode OnTimeOut() d'un objet IEvent() quand celui-ci a dépassé sa durée de validité, et appeler OnEvent() quand cet événement survient.

Dans la librairie, on ne peut pas savoir à quoi correspondra un événement. C'est pourquoi la classe IEvent est abstraite.

Pour l'utiliser, il faudra donc dériver une classe à partir de IEvent, implémentant les méthodes OnTimeOut() et OnEvent() et contenant des attributs. Par la suite, on ajoutera des événements par l'intermédiaire de AddEvent(), et le moteur se charge de gérer les événements indépendamment de l'implémentation des méthodes virtuelles de la classe IEvent.

3.16 Pointeurs, Références et Valeurs

Il est INTERDIT de prendre des références sur les types de base (int, char, etc.).

Le passage d'instances de classes à une fonction ou une méthode par valeur est interdit. On ne passera que des pointeurs ou des références à une instance de classe.

On utilisera donc, par ordre de préférence :

1. le passage par pointeur,
2. le passage par référence constante (si l'attribut const pour la classe a été prévu lors de la conception de celle-ci),
3. le passage par référence non-constante.

De manière générale, le choix entre pointeur et référence devra se faire comme suit :

- Si l'on désire garantir que l'on ne pointe pas sur NULL, alors on utilisera une référence.
- Si l'on désire avoir la possibilité de pointer sur NULL (pour indiquer que l'instance n'existe pas encore, par exemple), alors on utilisera un pointeur.
- Si l'on désire garder le pointeur après la sortie de la méthode… alors on utilisera un pointeur.

3.17 Chaînes de Caractères

Dans la plupart des cas, les opérations à effectuer sur les chaînes de caractères sont suffisamment simples pour que l'on utilise des tableaux de char terminés par un caractère zéro.

L'utilisation d'objets string est tolérée, notamment dans les cas où les performances ne s'en trouvent pas dégradées de façon significative, et lorsque les manipulations à effectuer sur la chaîne sont telles qu'elle rend le code plus simple et plus lisible.

3.18 Booléens

Le type bool est à utiliser obligatoirement pour toutes les variables qui ne peuvent prendre que deux valeurs (habituellement 0 et 1).

Le préfixe de type correspondant est f (cf. 2.2.2 Nommage des Variables).

3.19 Constructeurs / Destructeurs

Il est interdit de lever une exception dans un constructeur par recopie.

Il est souhaitable qu'un constructeur n'acquière de ressources que par des recopies et des allocations mémoire.
S'il doit y avoir d'autres initialisations plus complexes (ouverture d'une socket, ouverture d'un fichier, …), on utilisera une méthode Init().

Un destructeur ne devra renvoyer aucune exception.

Tout ce qui a été alloué dans le constructeur doit être libéré dans le destructeur, et tout ce que a été alloué dans une méthode de type Init() devra être libéré dans une méthode de type Exit().

Le destructeur aura pour tâche de « finir le ménage », surtout dans le cas où l'Exit() n'a pas été appelée.

3.20 Utilisation du mot-clé "const"

Il est primordial d'informer l'utilisateur d'une méthode qu'il est garanti que tel ou tel objet ne sera pas modifié par cette méthode (ou fonction).

La propriété const devra être utilisée systématiquement pour les pointeurs et les références, dès qu'un objet n'est pas censé être modifié par l'intermédiaire de ce pointeur / cette référence.
Si une méthode ne modifie pas l'objet auquel elle appartient, elle doit impérativement être déclarée avec le suffixe const. Voici un exemple tiré de l'ouvrage de B. Stroustrup « The C++ Programming Language » :

```
class CDate {
  int iDay, iMonth ;
public :
  int Day(void) const {return iDay ;}
  int Month(void) const {return iMonth ;}
  // ...
} ;
```

La présence du suffixe const après le prototype des fonctions Day() et Month() indique que ces fonctions ne modifient pas l'état d'un objet CDate.

L'attention des développeurs est attirée sur certaines confusions à éviter. Par exemple, les déclarations suivantes sont différentes :

```
void f(const MaClass * p);       // p ne peut etre utilise pour modifier l'objet
                                 // sur lequel il pointe
```

Et

```
void f(MaClass * const p);       // p ne peut etre modifie
```

En pratique, seule la première déclaration est utile et devra être utilisée.