Règles de codage C++

Révision	Référence	Date
01	L-DI-RDC-CPP	15/11/2012

Table des évolutions

Révision	Auteur	Date	Commentaire
01	mickael@leka	15/11/2012	Création du document

Table des matières

Introduction

Définition du document

Le présent document définit les règles de codage à adopter dans le cadre d'un développement logiciel dans le langage C++.

Ces règles sont basées sur des standards établis et acquis de diverses expériences individuelles, de sources différentes et de besoins différents.

Il y a plusieurs raison à l'introduction de ces règles de codage. La première est qu'elles permettent une harmonie au sein des projets de Leka et faciliteront l'utilisation, la relecture et la modification des programmes existants ou à venir.

Ce document ne contient aucune recommandation technique. Il spécifie juste le style de codage à adopter.

Pourquoi s'embêter avec du style de codage alors que certains IDE améliorent déjà la lisibilité du code, de manière visuel, avec de la coloration syntaxique, de la navigation par lien entre éléments, etc… ? Le code source du programme doit toujours être considéré en dehors de l'environnement de développement dans lequel il est composé et doit être écrit de manière à améliorer sa lisibilité quel que soit l'environnement utilisé. De cette manière, au sein d'une équipe, chacun peut utiliser les outils qu'il veut et qu'il aime de manière à programmer le plus efficacement et confortablement possible.

Définitions des règles

Les règles sont numérotées et seront présentées de la manière suivante :

n. Directive
`Exemple de code, si applicable.`
Motivation, raison d'être et informations additionnelles.

Importance des règles

Dans la partie de la directive, les termes **doit**, **devrait** et **peut** ont chacun une signification particulière. Une directive avec un **doit**, doit être appliquée, un **devrait** est une forte recommandation et un **peut** est une directive générale.

Règles générales

1. Toute non-conformité est autorisée si elle permet une meilleure lisibilité.

Le but principal de ces règles est d'améliorer la lisibilité et donc la compréhension, la maintenabilité et la qualité générale du code. Il n'est pas possible de couvrir tous les cas spécifiques dans ce document et le développeur se doit d'être flexible.

Conventions de nommage

Conventions générales de nommage

2. Un nom représentant un type doit être en casse mixte et doit commencer par une majuscule.

```
SuperClasse, Type, Objet
```

Pratique commune dans le développement en C++.

3. Une variable doit être en casse mixte et commencer par une minuscule.

```
superClasse, type, objet
```

Pratique commune dans le développement en C++. Facilite la distinction avec les types et résout potentiellement les collisions dans les déclarations comme Objet objet;

4. Une constante (énumération aussi) doit être entièrement en majuscule, les mots étant séparés par des tirets-bas.

```
MAX_VALUE, PI
```

Pratique commune dans le développement en C++. En règle générale, l'utilisation des constantes doit être minimisée. L'utilisation d'une méthode est préférée :

```
int getMaxValue() {
      return 10;
}
```

Cette forme permet d'unifier les interfaces des classes et d'implémenter dans le futur une méthode de calcul de la constante.

5. Un nom représentant une méthode ou une fonction doit être verbeux et écrit en casse mixte en commençant par une minuscule.

```
getValue(), computeSomething()
```

Pratique commune dans le développement en C++. Identique aux noms de variables mais les fonctions sont déjà distinguables de par leur forme spécifique.

6. Un nom représentant un espace de noms doit être entièrement en minuscule

```
leka::balle, common::image
```

Pratique commune dans le développement en C++.

7. Un nom représentant un template doit être une simple lettre majuscule.

```
template<class T> …
template<class K, class V> …
```

Pratique commune dans le développement en C++. Fait que les noms de template restent identifiables par rapport à tous les autres noms utilisés.

8. Les abréviations et les acronymes ne doivent pas être en majuscules lorsqu'ils sont utilisés en tant que nom.

```
exportHtmlSource();
openDvdPlayer();
```

Utiliser uniquement des majuscules pour le nom rentrerait en conflit avec la convention de nommage [3] des variables. Une variable de ce type devrait être nommé dVD ou encore hTML, etc… ce qui n'est pas très lisible. Un autre problème est quand deux de ces noms sont un derrière l'autre, la lisibilité en est sérieusement réduite.

9. Une variable globale doit être référencé en utilisant l'opérateur ::

```
::globalVariable
```

L'utilisation des variables globales est proscrite, dans ce cas, l'utilisation d'un pattern singleton est à considérer.

10. Un attribut privé d'une classe doit être préfixé par un tiret-bas.

```
class MyClass {
private :
      int _privateValue;
}
```

En dehors du nom et du type, la portée de la variable est importante. L'utilisation du tiret-bas en préfixe d'une variable permet de la distinguer d'une variable locale et

doit donc être considéré avec une autre attention par le développeur.

Un autre effet de ce préfixe est qu'il résout proprement les problèmes de trouver un nom raisonnable pour les setters et les constructeurs :

```
void setDepth(int depth) {
      _depth = depth ;
}
```

11. Une variable générique doit avoir le même nom que son type.

```
void setTopic(Topic * topic);
void connect(Database * database);
```

Réduit la complexité en réduisant le nombre de termes et noms utilisés. Cela rend aussi plus facile de déduire le type de la variable uniquement par son nom.

Si pour quelconque raison cette règle n'a pas l'air de bien s'appliquer, c'est que le nom du type de la variable est mal choisit.

Les variables non génériques ont un rôle. Elles peuvent souvent être nommées en combinant le rôle et le type :

```
Point startingPoint, centerPoint;
Name loginName;
```

12. Tous les noms doivent écrit en anglais.

```
fileName; // Pas nomFichier
```

L'anglais est la langue de référence en termes de développement.

13. Une variable ayant une grosse portée devrait avoir un long nom, une variable avec une faible portée peut avoir un nom court.

Les variables temporaires ou les indices sont bien mieux avec un nom court. Un programmeur lisant un telle variable devrait être capable d'assumer que sa valeur n'est pas utilisé en dehors de quelques lignes de code.

Les variables courtes utilisées pour les entiers sont *i*, *j*, *k*, *m*, *n* et pour les caractères *c* et *d*, *f* pour les flottants.

14. Le nom d'un objet est implicite et ne devrait pas être répété dans le nom d'un méthode.

```
line.getLength(); // NON : line.getLineLength();
```

Ce qui peut être naturel au moment de la déclaration de la classe, se voit être

superflu au moment de l'utilisation.

Conventions de nommage spécifiques

15. Les termes *get/set* doivent être utilisés quand l'accès à un attribut est direct.

```
person.getName();
person.setName(name);
image.getValue(x, y);
image.setValue(2, 3, value);
```

Pratique commune dans le développement en C++. En Java, cette convention est devenue standard.

16. Le terme *compute* peut être utilisé dans les méhode quand quelque chose est calculé.

```
valueSet->computeAverage();
matrix.computeInverse();
```

Donne immédiatement au lecteur l'idée que la méthode appelé est potentiellement longue à être exécutée.

17. Le terme *find* peut être utilisé quand quelque chose est recherché.

```
matrix.findMinElement();
map.find(key);
```

Donne immédiatement au lecteur l'idée que quelque chose est recherché.

18. Le terme *initialize* peut être utilisé là où un objet ou un concept est établi.

```
printer.initializeFontSet();
```

Le terme américain *initialize* est préféré plutôt que l'anglais *initialise* ou l'abréviation *init* à proscrire.

19. Une variable représentant un composant graphique doit être suffixé par le type du composant représenté.

```
exitButton, loginText, imageCanvas
```

Améliore la lisibilité en donnant au lecteur immédiatement l'idée de l'objet représenté par la variable.

20. Le pluriel doit être utilisé sur les noms utilisés pour représenter une collection

d'objets.

```
vector<Point> points;
int values[];
```

Améliore la lisibilité en spécifiant directement au lecteur le type de variable et les opérations pouvant être appliquées sur l'élément.

21. Le préfixe n devrait être utilisé pour une variable représentant un nombre d'objets.

```
nPoints, nLines
```

La notation est empruntée des mathématiques où la convention est établie pour indiquer un nombre d'objets.

22. Le préfixe *i* devrait être utilisé pour les variables représentant un numéro d'entité.

```
iPoint, iTable
```

Cela fait de la variable un itérateur nommé.

23. Un itérateur devrait être appellé i, j, k, etc…

```
for (int i = 0; i < nTables); i++) {
      :
}
for (vector<MyClass>::iterator i = list.begin(); i != list.end(); i++) {
      Element element = *i;
      ...
}
```

Cette notation est empruntée des Mathématiques où la convention est établie pour signaler un itérateur.

Les variables j, k, etc… ne devrait être utilisées que pour les boucles imbriquées.

24. Le préfixe *is* devrait être utiliser pour les variables et les méthodes booléene.

```
isSet, isVisible, isOpen
```

Pratique commune dans le développement en C++.

L'utilisation du préfixe *is* résout les problèmes de mauvais choix de nom pour les booléns et le développeur est donc forcé de choisir des noms plus adaptés. Par exemple : status, flag => isStatus, isFlag ne conviennent pas.

Il y a certaines alternatives qui vont mieux pour certaines situations :

- `bool hasLicense();`
- `bool canEvaluate();`
- `bool shouldSort();`

25. Les noms complémentaires doivent être utilisés pour les opérations complémentaires.

```
get/set, add/remove, create/destroy, start/stop, insert/delete, old/new,
begin/end, first/last, etc…
```

La symétrie réduit la compléxité.

26. Les abréviations dans les noms devrait être proscrite.

```
computeAverage(); // NON : compAvg
```

Il y a deux types de mots à considérer. Tout d'abord les mots communs que l'on trouve dans un dictionnaire. Ces derniers ne doivent pas être abréviés. Par exemple, ne jamais écrire :

- `cmd` au lieu de `command`
- `cp` au lieu de `copy`
- `pt` au lieu de `point`

Ensuite, il y a des phrases spécifiques à certains domaines qui sont plus naturellement connues sous forme d'abréviations / d'acronymes. Ces phrases doivent rester abrégées. Ne jamais écrire :

- `hypertextMarkupLanguage` au lieu de `html`
- `centralProcessingUnit` au lieu de `cpu`
- etc…

27. Nommer spécifiquement les pointeurs devrait être proscrit.

```
Line * line; // NON : Line * pLine; ou Line * LinePtr;
```

Beaucoup de variables en C/C++ sont des pointeurs, donc une règle comme celle-ci devient vite difficile à suivre. De plus, les objets en C++ sont souvent des types transverses dont leur implémentation doit être ignorée par le développeur. Le nom doit représenter le type uniquement quand le type d'objet a une signification spéciale.

28. Les booléen nommés négativement doivent être proscrit.

```
bool isError; // NON isNoError
bool isFound; // NON isNotFound
```

Les problèmes arrivent lorsqu'un tel nom vient à être utilisé avec l'opérateur logique non, ce qui donne un résultat doublement négatif. La signification de `!isNotFound` est beaucoup moins compréhensible.

29. Les constantes énumérées doivent toujours référer à leur type énuméré.

```
Color::RED, Dimension::2D
```

Cette règle permet de donner plus d'informations sur la provenance de la déclaration.

De plus, on notera que le type énuméré doit être au singulier. Un nom au pluriel comme `enum Colors { … }` peut paraitre bon, mais devient vite sale à l'utilisation.

30. Les classes d'exceptions doivent être suffixé par *Exception*.

```
class AccessException {
      :
}
```

Les classes d'exceptions ne font pas réellement partie du design principal du programme et les nommer de cette manière permet de les distinguer des autres classes.

31. Les fonctions devraient être nommées en fonction de ce qu'elles renvoient et les procédures en fonction de ce qu'elles font.

Augmente la lisibilité. Rend plus clair ce que l'élément doit faire et spécialement tous ce qu'il n'est pas supposé faire.

Les fichiers

Les fichiers sources

32. Les headers C++ doivent avoir l'extension *.h* (préféré) ou *.hpp*. Les fichiers sources peuvent avoir l'extension .cpp (recommandé), .c, .cc ou .c++.

```
MyClass.cpp, MyClass.h
```

Ce sont les extensions acceptées par le standard C++.

33. Une classe doit être déclaré dans un fichier d'en-tête et définie dans un fichier source dont les deux ont le nom de la classe.

```
MyClass.h, MyClass.cpp
```

Rend plus facile la recherche de fichier d'une certaine classe. La seule exception est les templates qui doivent être à la fois déclarés et définies dans le fichier *.h*.

34. Toutes les définitions devraient résider dans le fichier source.

```
class MyClass {
public:
      int getValue() { return _value; } // NON !
      ...

private:
      int _value;
}
```

Les fichiers d'en-tête doivent déclarer une interface et le fichier source l'implémenter. Quand une implémentation est recherchée, le développeur doit toujours savoir qu'elle se trouve dans le fichier source.

35. Les instructions tenant sur plusieurs lignes doivent être lisible.

```
function (param1, param2,
          param3);
```

Il est difficile de ressortir une règle générale. Pour ces cas-là se référencer à la règle [1].

En règle générale :

- Sauter une ligne après une virgule.
- Sauter une ligne après un opérateur.
- Aligner les nouvelles lignes par rapport aux précédentes.

Inclusion des fichiers et instructions d'inclusion

36. Les fichiers d'en-têtes doivent avoir une protection contre la redéfinition.

```
#ifndef __COM_LEKA_MODULE_CLASSNAME_H_
#define __COM_LEKA_MODULE_CLASSNAME_H_
 :
#endif // __COM_LEKA_MODULE_CLASSNAME_H_
```

Cette protection permet d'éviter les erreurs de compilation. La convention de nommage reflète l'espace de nom pour éviter les conflits.

37. Les inclusions de fichiers doivent être triées et groupées. Triées par leur position hiérarchique avec l'inclusion des fichiers de bas niveau en premier.

```
#include <fstream>
#include <iomanip>

#include <qt/qbutton.h>
#include <qt/qtextfield.h>

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/ui/MainWindow.h"
```

En plus de montre au lecteur les fichiers individuels inclus, cela donne aussi un aperçu immédiat des fichiers impliqués.

38. L'inclusion des fichiers doit se faire uniquement au début du fichier.

Pratique commune qui évite les effets secondaires d'une inclusion cachée au fond du code source.

Déclarations

Types

39. Les types qui sont local à un seul fichier peuvent être déclarés dans ce fichier.

Met en œuvre le cache d'information.

40. Les parties d'une classe doivent être ordonnées par *public*, *protected* et *private*. Toutes les sections doivent être explicitement déclarées. Les sections non-applicables ne doivent pas apparaitre.

L'ordre est : le plus *plublic* en premier. De cette manière les gens voulant uniquement utiliser la classe peuvent s'arrêter de lire quand ils atteignent la section *protected / private*.

Variables

41. Les variables doivent être déclarés là où elles sont initialisées.

Permet à ce que les variables soient tout le temps valides. Parfois, il est imposssible de les initialiser à une valeur correct là où elles sont déclarées :

```
int x, y, z;
getCenter(&x, &y, &z);
```

Dans ce cas, elles doivent rester non-initialisées plutôt que d'être initialisées à une valeur incohérente.

42. Une variable ne doit jamais avoir deux significations différentes.

Augmente la lisibilité en assurant que les concepts sont représentés de manière unique. Réduit la chance d'avoir des effets indésirables.

43. L'utilisation des variables globales doit être proscrite.

En C++, il n'y a aucune raison pour que les variables globales soient utilisées. C'est aussi vrai pour les fonctions globales ou les variables statiques avec portance dans un fichier.

44. Les attributs de classe ne devraient jamais être déclarés publique.

Le concept C++ de la dissimulation de l'information et de l'encapsulation est réduit à néant avec les variables publiques. Il vaut mieux utiliser des variables privées avec des accesseurs. L'exception est lors de l'utilisation d'une classe comme d'une structure de données. Dans ce cas, il est approprié de rendre les attributs publics.

Notons qu'en C++, les *structs* ont été conservées uniquement par raison de compatibilité et doivent être proscrites.

45. Les tests implicite pour l'égalité pour 0 ne doivent pas être utilisés autrement que par des booléen ou des pointeurs.

```
if (nLines != 0) // NON if (nLines)
if (value != 0.0) // NON if (value)
```

En C++, il n'est pas nécessaire que la valeur 0 des entiers et des flottants soit implémenté comme un 0 binaire. De plus, l'utilisation explicite du test permet de voir directement un aperçu du type de la valeur testée.

46. Les variables doivent être déclarées avec la plus petite portée possible.

Conserver les opérations sur les variables dans une petite portée rend plus facile la perception des effets ou des effets indésirables d'une variable.

Boucles

47. Seul les déclarations de contrôle de la boucle *for()* doivent être déclarés dans sa construction.

```
sum = 0 ;                                   // NON : for (i = 0, sum = 0; i < 100 …
for (i = 0; i < 100; i++) {
      sum += value[i];
}
```

Augmente la lisibilité et la maintenabilité. Fait la distinction entre ce qui contrôle et ce que contient la boucle.

48. Les variables de boucles doivent être initialisés immédiatement avant la boucle.

```
isDone = false;          // NON bool isDone = false
while ( !isDone ) {      //       :
      :                  //     while ( !isDone ) {
}
```

49. La forme *while(true)* doit être utilisé pour les boucles inifinies.

```
while (true) {}
// NON for (;;) {}
// NON while (1) {}
```

Tester la valeur 1 n'est pas nécessaire en plus de ne pas être significative. La forme *for(;;)* n'est pas très lisible et ne montre pas que c'est une boucle infinie.

Conditions

50. Les conditions complexes doivent être proscrites. Introduire des variables booléennes pour simplifier l'évaluation.

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
 :
```

```
}
// NOT: if ((elementNo < 0) || (elementNo > maxElement)|| elementNo ==
lastElement) {
 :
}
```

En assignant des valeurs booléennes aux expressions, on documente automatiquement le programme. La construction sera plus simple à lire, à débugger et à maintenir.

51. Le cas nominal doit être placé dans la partie avec le *if*, l'exception dans la partie avec le *else*.

```
bool isOk = readFile(fileName);
if (isOk) {
      :
}
else {
      :
}
```

Fait en sorte que l'exception ne vienne pas cacher le chemin normal de l'exécution. Important pour la lisibilité et la performance.

52. L'instruction conditionnelle doit être écrite sur une nouvelle ligne.

```
if (isDone)              // NON if (isDone) doCleanUp();
      doCleanUp();
```

Utile lors du débogage. Si une seule ligne est utilisée, on ne sait pas si le test réalisé est vrai ou faux.

53. Les exécutions d'instructions dans les conditions doivent être proscrites.

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) {
      :
}
// NON
if (!(fileHandle = open(fileName, "w"))) {
      :
}
```

Difficile à lire pour les développeurs encore nouveau au C++.

Divers

54. L'utilisation des nombre magiques autre que 0 et 1 dans le code doivent être déclarés en tant que constante nommées.

Si le nombre n'a pas de signification évidente, la lisibilité est augmenté par l'introduction d'une constante qui le définit.

55. Les nombres flottants doivent toujours être écrit avec une séparation des nombres décimaux et au minimum un nombre décimal.

```
float total = 0.0f;   // NOT: float total = 0f;
float speed = 3.0e8f; // NOT: float speed = 3e8f;
float sum;
 :
sum = (a + b) * 10.0f;
```

Augmente la différenciation de leur nature par rapport aux entiers. Mathématiquement parlant, ce sont deux modèles complètement différents et non-compatibles.

De plus, cela met en valeur le type de la variable *sum* à un point où elle n'est pas forcément évidente.

56. Les fonctions doivent toujours avoir le type de la valeur de retour spécifié.

```
int getValue() {   // NOT: getValue()
     :
}
```

Si le type n'est pas explicite, le C++ considère que le type de retour par défaut est un entier pour les fonctions.

57. Goto ne doit jamais être utilisé

La déclaration *goto* viole l'idée d'un code structuré.

Disposition et commentaires

Disposition

58. L'indentation basic doit être une tabulation.

Par la suite, la taille de l'indentation peut être choisie par chacun à son affichage pour une meilleure lisibilité.

59. Le style de code doit être celui de Kernighan and Ritchie ou le Compact control readability style.

Permet une meilleure lecture du code. Les définitions des styles sont disponibles sur Wikipédia : http://en.wikipedia.org/wiki/Indent_style

Commentaires

60. Le code délicat ne doit pas être commenté mais réécrit !

En général, l'utilisation des commentaires doit être minimisée en auto-documentant le code par le choix de noms appropriés et d'une structure logique explicite.

61. Les commentaires doivent être inclus relativement à leur position dans le code.

```
while (true) {                  // NON :    while (true) {
      // Do something                        // Do something
      something();                                 something();
}                                           }
```

62. Les en-têtes de classe et de méthodes doivent suivre la convention de JavaDoc.

La standardisation de la documentation des méthodes et des classes en Java est bien plus mature qu'en C++. C'est dû au fait d'outils disponibles pour créer une documentation automatiquement, comme JavaDoc.

Il y a aussi des outils similaires à JavaDoc disponibles pour le C++. Ces outils suivent le même principe de syntax que la JavaDoc. Par exemple : Doc++ ou Doxygen.