Conventions de codage

Révisées le 18 juin 2003

Sommaire

1. Introduction

Les différents points abordés sont de deux types : les règles et les recommandations. Les règles expriment les principes devant être absolument respectés, alors que les recommandations expriment des principes optionnels appliqués ou pas au bon vouloir de chacun. Ces différents points sont ensuite regroupés en parties qui vous permettront de mieux retrouver l'information.

2. Sources et fichiers

2.1. Généralités

Règle 1 : Langue préférée

Quand il s'agit de code source (noms de fichiers, identifiants, commentaires, documentation, ...), la langue préférée est l'anglais.

Toutefois, et ceci exclusivement dans les commentaires et dans la documentation, si l'utilisation de l'anglais poserait des problèmes pour vous exprimer, dans un soucis de clarté vous pouvez envisager l'emploi de votre langue natale.

Règle 2 : Noms des fichiers

Le nom d'un fichier est basé, dans la mesure du possible, sur le nom de la classe qui y est déclarée ou définie, en respectant la casse de celui-ci.

Règle 3 : Extensions des fichiers

Les fichiers d'en-têtes ont l'extension « .h ». Les fichiers d'implémentation ont l'extension « .cpp ». Les fichiers d'implémentation de classes « templates » ont l'extension « .hxx ».

Règle 4 : Organisation des fichiers

Vous devez refléter dans l'organisation de vos fichiers sources l'arborescence des namespaces (cf page 21) de votre code.

Recommandation 1 : Une seule classe par fichier

Il est généralement mieux de ne déclarer (ou implémenter) qu'une seule classe par fichier, afin de mieux structurer le code. Certains exprimeront à raison une objection à cette recommandation, en argumentant que de très petites classes peuvent facilement être déclarées (ou implémentées) dans le même fichier.

2.2. En-têtes et inclusions

Règle 5 : Bon type d'inclusion

Utilisez la directive d'inclusion qui correspond au contexte dans lequel vous importez une en-tête. `#include ""` servira à l'importation d'en-têtes provenant d'un même module, alors que `#include <>` servira à l'importation d'en-têtes provenant d'autres modules.

Règle 6 : Bon chemin d'inclusion

Un bon chemin d'inclusion n'est pas un chemin absolu dépendant d'un système de fichiers. Un bon chemin d'inclusion respecte la casse des noms de fichiers et de répertoires (certains systèmes y étant sensibles). Un bon chemin d'inclusion utilise le caractère '/' comme séparateur.

Règle 7 : Protection contre les inclusions multiples

Vous devez protéger vos en-têtes contre les inclusions multiples. Pour cela, utilisez une technique supportée par tout compilateur (`#pragma once` n'est supporté que par les compilateurs Microsoft), se basant les directives standards du pré-compilateur qui sont `#ifndef` et `#define`. Utilisez le nom du fichier ainsi que la hiérarchie de namespace dans la balise de protection, afin d'éviter tout conflit avec un

fichier ayant le même nom mais ne se trouvant pas dans le même namespace.

Recommandation 2 : En-têtes « clef en main »

Ne perdez jamais de vue qu'un fichier d'en-tête doit être fourni « clef en main ». Il doit être possible d'inclure un fichier d'en-tête n'importe quand, sans avoir à faire d'autres inclusions préalables afin de pouvoir compiler. Ainsi, n'oubliez pas d'inclure dans vos en-têtes toute autre en-tête et déclarations par anticipation nécessaires à la compilation de votre code.

Recommandation 3 : Minimisation des inclusions

Minimiser autant que possible les inclusions dans un fichier d'en-tête en vous servant de déclarations par anticipation. Ceci permet de réduire les dépendances entre les fichiers et réduit ainsi les temps de compilation.

Recommandation 4 : Trier les inclusions

Vous pouvez éventuellement trier les inclusions dans l'ordre alpha-numérique du chemin. Les inclusions se trouverons alors regroupées par module.

Exemple 1 : Fichier d'en-tête

```
#ifndef A_B_C_SAMPLE_H // Preamble protecting against multiple
#define A_B_C_SAMPLE_H // inclusions.


class OtherClass;  // Forward declaration,
                   // instead of including the relevant header.


namespace a
{
namespace b
{
namespace c
{


class Sample
{
    ...

private:

    OtherClass*   m_membre; // Usage of the forward declared class.
};


} // namespace c
} // namespace b
} // namespace a


#endif // A_B_C_SAMPLE_H
```

Exemple 2 : Inclusions triées

```
#include <libSample/first.h>
#include <libSample/second.h>
#include <libSample/subModule/first.h>
#include <libSample/subModule/second.h>
#include <map>
#include <vector>
```

2.3. Documentation

Règle 8 : Comment documenter ?

Vous documenterez votre code avec Doxygen, un outil de génération automatique de documentation. Vous vous conformerez donc aux spécificités de cet outil.

Règle 9 : Où documenter ?

Toute documentation utile à l'utilisateur doit se trouver dans les fichiers d'en-tête. Ainsi tout utilisateur d'un module trouvera en une fois la déclaration d'une classe et sa documentation. Dans les fichiers d'implémentation, la documentation donnera des détails supplémentaires et précis sur la réalisation des algorithmes.

De plus, toute documentation devra se trouver juste avant l'entité à laquelle il est fait référence, ceci afin de faciliter la recherche d'information au sein du code.

Recommandation 5 : Blocs de documentation légers

Dans un bloc de commentaire Doxygen, ne placez que les balises strictement nécessaires. Ceci évitera de surcharger la documentation et la rendra donc plus lisible pour un utilisateur. De plus, les balises vides se retrouveront dans la documentation finalement générée, où elles seront toujours aussi peu pertinentes.

Ne vous sentez pas non plus obligés de sectionner un bloc de documentation en utilisant des lignes de séparations autres que des lignes vides (par oppositions aux lignes d'astérisques).

Enfin, lors de la documentation d'une fonction, ne reprenez pas le type des paramètres. D'une part, cette information est inutile du fait que le profil de la fonction n'est pas loin et qu'il est donc facile de retrouver cette information. D'autre part, en agissant ainsi, la génération de la documentation est perturbée car Doxygen considère le premier mot suivant la directive `@param` comme étant le nom du paramètre.

Recommandation 6 : Documentation structurée

Doxygen se charge de fournir une certaine structure lors de la génération de la documentation. Toutefois, au sein d'une entité documentée de grande taille, il est souvent judicieux d'utiliser les possibilités de groupage (`@name`) de Doxygen. Vous construirez ainsi une vue logique des interfaces de classe.

Recommandation 7 : Se documenter sur Doxygen

N'omettez pas de consulter la documentation de Doxygen. Les possibilités offertes par cet outil sont assez nombreuses, aussi il est souvent bon de bien se renseigner sur les balises disponibles et leur interprétation.

Exemple 3 : Documentation malheureuse

```
/**
 * @brief          A very short description.
 *
 * A longer description, giving more details about the documented piece
 * of code.
 **********************************************************************
 * @param
 **********************************************************************
 * @return
 **********************************************************************
 * @exception
 **********************************************************************
 * @todo
 **********************************************************************/
```

Exemple 4 : Documentation heureuse

```
/**
 * @brief          A very short description.
 *
 * A longer description, giving more details about the documented piece
 * of code.
 */
```

Exemple 5 : Documentation de fonctions

```
class Sample
{
    /**
     * Retrieve the thing.
     *
     * @return    The thing value.
     */
    const std::string& GetThing( void ) const;

    /**
     * @brief Set the thing.
     *
     * @param     thing     :  The new thing.
     */
    void SetThing( const std::string& thing );
};
```

3. Les identifiants

3.1. Généralités

Règle 10 : Composition et casse

Les identifiants peuvent être composé par concaténation de plusieurs mots. Afin de séparer chaque mot dans un identifiant, mettez la première lettre de chaque mot en majuscule, les autres étant en minuscule.

Règle 11 : Caractères interdits

Ne commencez pas un identifiant par un ou plusieurs caractères trait bas « _ ». N'utilisez pas de caractère trait bas « _ » dans les identifiants. Notez que d'après la norme ANSI-C, le double trait bas « __ » est réservé au compilateur. Notez aussi que le trait bas simple « _ » est parfois utilisé en préfixe dans les identifiants internes de certaines librairies.

Comme toute bonne règle, vous trouverez plus loin plusieurs exceptions qui imposeront l'utilisation du caractères trait bas « _ » dans certains cas.

Règle 12 : Pertinence

Utilisez toujours des identifiants qui ont un sens explicite. N'utilisez pas d'abréviations. Il est préférable que les identifiants soient longs et compréhensibles que courts et indéchiffrables.

3.2. Les types

Règle 13 : Types définis par l'utilisateur

Les identifiants de types définis par l'utilisateur (classes, structures, types énumérés et *typedef*) commencent toujours par une majuscule.

Règle 14 : Types scalaires (ou alias sur les types natifs)

Les identifiants de types scalaires (types natifs du langage et alias définis dans les librairies) sont écrits exclusivement en lettres minuscules.

Règle 15 : Paramètres « templates »

Les paramètres « templates » qui sont des noms de types suivront les règles relatives au identifiants des types utilisateurs.

Il est recommandé d'utiliser des identifiants relativement courts pour les paramètres « templates » afin d'augmenter la lisibilité du code. Il ne s'agit pas pour autant d'utiliser des identifiants se résumant à un seul caractère.

Exemple 6 : Types utilisateurs

```
class SampleClass;

struct SampleStruct;

typedef SampleClass SampleClassAlias;

enum SampleEnumeration
{
    ...
};
```

Exemple 7 : Types scalaires

```
typedef unsigned char  uint8;
typedef char           int8;
typedef unsigned int   uint32;
typedef int            int32;
```

Exemple 8 : Paramètres « templates »

```
template< class Key, class Value > class SampleClass
{
    ...
};
```

3.3. Les variables et paramètres

Règle 16 : Variables et paramètres

Les identifiants de variables et de paramètres commencent toujours par une minuscule.

Règle 17 : Variables membres

Les identifiants de variables membres (d'instance ou de classe) sont toujours préfixées par `m_`. Toutefois, cette règle ne s'applique pas aux membres exposés publiquement par une structure simple..

Recommandation 8 : Informations de type

Les identifiants de variables et de paramètres scalaires peuvent éventuellement recevoir une information supplémentaire de type, utilisant une forme abrégée de l'identifiant du type. Le tableau ci-dessous présente les types scalaires courants et leurs abréviations.

Type	***Abréviation***	***Description***
bool	b	Valeur booléenne.
int8	i8	Entier signé de 8 bits.
uint8	ui8	Entier non signé de 8 bits.
int16	i16	Entier signé de 16 bits
uint16	ui16	Entier non signé de 16 bits.
int32	i32	Entier signé de 32 bits.
uint32	ui32	Entier non signé de 32 bits.

Type	*Abréviation*	*Description*
float	f	Flottant en précision simple.
double	d	Flottant en précision double.
*	p	Pointeur.
std::string	str	Chaîne de caractères STL ANSI.
std::wstring	wstr	Chaîne de caractères STL Unicode.

Si l'information de type est placée en préfixe, la première lettre du nom de la variable sera en majuscule, afin de bien le mettre en valeur.

Si l'information de type est placée en suffixe, elle sera séparée du nom de la variable par un seul caractère « _ », afin de mettre le nom en valeur.

Notez enfin que certaines API système (comme Win32) définissent leur propre préfixes de type. Il est évidement possible et même recommandé d'utiliser ces préfixes lors de développements utilisant directement ces API.

Recommandation 9 : Cohérence

Dans la mesure du possible, la cohérence des noms de variables devra être conservée dans chaque module (libraire ou programme). Ainsi, si un préfixe ou un suffixe de type est utilisée dans le nom des variables, tout nom de variable de ce module devra se conformer à ce choix.

Exemple 9 : Variables et paramètres

```
const float myAverage( const float* pValues, const int count )
{
    int      currentIndex;
    float    sum;

    for( currentIndex = 0; currentIndex < count; currentIndex++ )
    {
         sum += pValues[ currentIndex ];
    }

    return sum / count;
}
```

Exemple 10 : Variables membres

```
class SampleClass
{
private:
    int      m_identifier;
    float    m_value;
};

struct DataCollection
{
    char     firstMembre;
    short    secondMembre;
    int      thirdMembre;
    long int fourthMembre;
    float    fifthMembre;
    double   sixthMembre;
};
```

3.4. Divers

Règle 18 : Etiquettes de types énumérés

Les étiquettes de types énumérés ne sont une exception aux règles énoncées jusqu'à présent. Elles doivent être rédigées exclusivement en lettres majuscules, chaque mot du nom étant séparé par un caractère '_'.

Exemple 11 : Enumération et étiquettes

```
enum SampleEnumeration
{
    FIRST_LABEL,
    SECOND_LABEL,
    THIRD_LABEL
};
```

4. Style

Cette partie traite les aspects formels du code source. Elle établit des règles et des recommandations relatives à la mise en forme de nombreux éléments du langage.

4.1. Généralités

Règle 19 : Tabulations pour les indentations

Lors de mise en forme du code, les indentations sont réalisées à l'aide de tabulations équivalantes à quatre caractères.

Recommandation 10 : Fonte pour la visualisation du code

Il est fortement recommandé d'utiliser une fonte à largeur fixe, pour une meilleur visualisation du code (alignement des mots à la lettre près, ...)

4.2. Les blocs

Règle 20 : Mise en forme des blocs

Aucun bloc ne devra être écrit sur une seule ligne, mais devra respecter la mise en forme, décrite ci-après.

Les lignes de code se trouvant à l'intérieur d'un bloc doivent être indentées d'une colonne vers la droite par rapport à la colonne de départ du bloc. L'accolade fermante d'un bloc doit se trouver obligatoirement seule sur une ligne, alignée sur la colonne de départ du bloc.

Toutefois, pour les accesseurs d'une classe définis lors de la déclaration, il est possible de placer le bloc sur une seule ligne, à la suite du prototype de la fonction. Un accesseur devra toujours se limiter au simple renvoi ou à la simple mise à jour d'une variable membre.

Règle 21 : Blocs de namespaces

Les corps de namespaces font exceptions à la règle d'indentation précédente, ceci pour des soucis de lisibilité et de maintenance du code source. En effet, au delà d'une profondeur de niveau un, la gestion des indentations deviendrait trop compliquée à gérer.

Règle 22 : Utilisation facultative de blocs

Quand un bloc à la suite d'une structure de contrôle (`if`, `else`, `for`, `while`) ne comprend qu'une seule instruction, il est possible de se passer du bloc. Pour se faire, vous devrez obligatoirement placer cette instruction sur la ligne de la structure de contrôle.

Recommandation 11 : Position de l'accolade ouvrante

L'accolade ouvrante d'un bloc se trouve de préférence seule sur un ligne, alignée sur la colonne de l'accolade fermante.

Exemple 12 : Accesseurs définis lors de la déclaration

```
class Sample
{
public:
    const int getValue( void ) const      { return m_value; }
    void      setValue( const int value ) { m_value = value; }

private:
    int m_value;
};
```

Exemple 13 : Mise en forme de namespaces

```
namespace a
{
namespace b
{
namespace c
{

...

} // namespace c
} // namespace b
} // namespace a
```

Exemple 14 : Mise en forme de structures de contrôle

```
void function( void )
{
    if( condition )
    {
        ...
    }
    else if( condition )
    {
        ...
    }

    for( ... ) SingleStatement( ... );
}
```

4.3. Les déclarations de classes

Dans le concept de classe, la notion d'encapsulation est très importante. Le plus grand nombre de mécanismes doit être caché pour l'utilisateur final. La complexité est alors placée à l'intérieur de l'objet et non pas à l'extérieur. La mise en forme de la déclaration de classes prend en compte ces considérations et propose une disposition qui rend l'information pertinente accessible pour les utilisateurs au plus vite.

Règle 23 : Trois sections pour le contrôle de la portée

N'utilisez pas plus d'une section `public`, `protected` et `private`, afin de mieux contrôler la portée des membres.

Règle 24 : Ordre des sections de contrôle de la portée

Mettez la section `public` en tête, elle concerne le plus grand nombre de personnes. Placez ensuite la section `protected`, elle concerne un groupe de personnes plus averties qui sont amenées à

implémenter les classes dérivées. La section `private` sera la dernière du faite que seuls les mainteneurs de la classe seront intéressés.

Règle 25 : Cas des classes « templates »

Pour les classes templates, la déclaration et la définition doivent être accessibles simultanément. Toutefois, vous ne définirez jamais les fonctions membres d'une classe « template » au sein de sa déclaration, mais après cette dernière. De cette façon la déclaration de son interface n'est pas polluée par les soucis et le code de l'implémentation.

Il est néanmoins possible de définir une fonction membre au sein de la déclaration d'une classe « template » quand il s'agit d'un accesseur (cf *Mise en forme des blocs*, page 10).

Recommandation 12 : Grouper les membres dans chaque section

Dans chacune des trois sections d'une déclaration de classe, vous pouvez grouper les membres par types : définitions de types, constructeurs et destructeur, opérateurs d'affectation, variables et enfin fonctions.

Recommandation 13 : Séparer la déclaration et l'implémentation des classes « templates »

Il est tout à fait possible de séparer la déclaration et l'implémentation d'une classe template. Pour ce faire, la déclaration se trouvera dans un fichier d'en-tête classique et l'implémentation se trouvera dans un autre fichier qui aura comme extension « .hxx » (cf *Extensions des fichiers*, page 2). Enfin, à la fin du fichier d'en-tête, vous inclurez le fichier d'implémentation.

Exemple 15 : Déclaration de classe non « template »

```
class Sample
{
public:

    // Constructors & destructor.
    Sample( void );
    ~Sample( void );

    // Accessors
    const int getSomething( void ) const;
    void      setSomething( const int something );

private:

    // Type definitions.
    typedef std::vector< std::string > VectorString;

    // Member variables.
    int   m_something;

    // Member functions.
    void doSomething( void );
};
```

Exemple 16 : Déclaration d'une classe « template »

```
// Template class declaration.

template< typename Param >
class Sample
{
public:
    void      doSomething ( void );
    const int getSomething( void ) const { return m_something; }

private:
    int m_something;
};


// Template class definition.

template< class Param >
void Sample< Param >::doSomething( void )
{
    ...
}
```

4.4. Les listes d'initialiseurs

Règle 26 : Chacun à sa place

Placez chaque initialiseur seul sur une ligne terminée le cas échéant par une virgule. Cette ligne est décalée d'une colonne vers la droite par rapport au nom du constructeur.

Pour les longues listes d'initialiseurs, il est également possible de regrouper plusieurs initialiseurs qui ont un lien (composantes de coordonnées, ...) sur une même ligne.

Règle 27 : Constructeurs des classes de bases

Placez toujours les constructeurs des classes de base en tête de la liste.

Règle 28 : Utilisation maximale

Vous utiliserez autant que possible les initialiseurs afin d'avoir des constructeurs plus homogènes (cf Initialisation des membres, page 19).

Recommandation 14 : Listes d'initialiseurs pour les extrémistes

Afin d'améliorer la présentation de listes d'initialiseurs, vous pouvez aligner les initialiseurs sur une colonne et leur liste d'argument sur une autre.

Exemple 17 : Liste d'initialiseurs

```
SampleClass::SampleClass( const std::string& name, const int value )
:   BaseClassOne  ( name ),
    BaseClassTwo  ( name ),
    m_value       ( value ),
    m_misc        ( 10 )
{}
```

4.5. Les fonctions

Règle 29 : Type de retour

Préciser toujours le type de retour dans la déclaration d'une fonction. Utilisez le mot clé `void` quand celle-ci ne retourne aucune valeur.

La plupart du temps, le compilateur émet un avertissement quand une fonction qui ne peut pas être un constructeur n'a pas de type de retour.

Règle 30 : Listes de paramètres vide

Quand une fonction n'accepte aucun paramètre, plutôt que d'écrire une liste vide, utilisez le mot clé `void`.

Règle 31 : Aucune expression en argument

Lors du passage d'arguments, ne placez en aucun cas d'expressions arithmétiques ou d'affectations. Effectuez l'évaluation de celles-ci (dans des variables intermédiaires si nécessaire) préalablement à l'appel de la fonction voulue. Ceci fournira un code plus compréhensible et permettra de visualiser la valeur des expressions lors de l'appel de la fonction au cours d'un débogage.

Cette règle peut toutefois être tempérée pour les expressions simples (n'effectuant qu'une seule opération qui n'est pas une affectation).

Règle 32 : Longues listes de paramètres (ou d'arguments)

Lors de la déclaration (respectivement de l'appel) de fonctions, les listes de paramètres (respectivement d'arguments) peuvent être longues. Vous devrez alors les éclater afin de rendre le code plus lisible.

Pour ce faire, chaque paramètre (respectivement argument) sera placé seul sur une ligne, la parenthèse ouvrante se trouvant sur la même ligne que le nom de la fonction et la parenthèse fermante se trouvant sur la ligne du dernier paramètre (respectivement argument). Alignez chaque paramètre (argument) sur une même colonne, décalée une fois vers la droite par rapport au nom de la fonction.

Recommandation 15 : Groupage de paramètres (d'arguments)

Dans le cas de longues listes de paramètres (d'arguments), il est également possible de faire un groupage logique de ces derniers (coordonnées, composantes de couleur, ...) en plaçant plusieurs paramètres (arguments) sur une même ligne.

Recommandation 16 : Liste de paramètres orphelines

Lors d'un appel de fonction, il est recommandé de ne pas séparer le nom de la fonction de la liste de paramètres.

Exemple 18 : Appel de fonction et expressions

```
// WRONG !
void badCall( void )
{
    int  i;
    int  j;
    int  k;
    int  l;

    ...

    // Do something !
    doDomething( (i + j) * (k + 500), l + 1 );
}


// GOOD !
void goodCall( void )
{
    int  i;
    int  j;
    int  k;
    int  l;

    ...

    // Do something !
    int  expressionValue = (i + j) * (k + 500);

    doDomething( expressionValue, l + 1 );
}
```

Exemple 19 : Déclaration (appel) de fonction avec une longue liste de paramètres (d'arguments)

```
void Sample::DoSomething(
    const std::string& strName,
    const int32        i32Value,
    const float        fAnotherValue,
    const float        fLastValue )
{
    DoAnotherThing(
         strName,
         i32Value,
         fAnotherValue,
         fLastValue );
}
```

4.6. Divers

Règle 33 : Enumérations

Mettez chaque étiquette d'une énumération seule sur une ligne, terminée éventuellement par une virgule. Quand vous assignez explicitement une valeur aux étiquettes, alignez ces valeurs sur une même colonne, l'opérateur d'affectation servant de référence pour cet alignement.

Règle 34 : Longues expressions

D'une manière générale, vous devrez évitez d'écrire de trop longues expressions. La plupart du temps, une expression est considérée comme étant trop longue quand elle ne peut pas être visualisée entièrement

à l'écran sans défilement.

Vous pouvez simplifier de telles expressions en calculant, dans des variables, des résultats intermédiaires. Il est aussi possible de répartir de telles expressions sur plusieurs lignes.

Recommandation 17 : Aérer le code

Il est fortement recommandé de placer un espace après toute virgule (comme dans les listes de paramètres).

Il est fortement recommandé d'aérer un minimum les expressions afin de les rendre plus lisibles.

Exemple 20 : Enumération

```
enum OpenFlag
{
    OPEN_SHARE_READ    = 1,
    OPEN_SHARE_WRITE   = 2,
    OPEN_EXISTING      = 4,
};
```

Exemple 21 : Expression longue

```
if( ('0' <= input && input <= '9' ) ||
    ('a' <= input && input <= 'z' ) ||
    ('A' <= input && input <= 'Z' ) )
{
    ...
}
```

Exemple 22 : Aération du code

```
// Parameter list sample.
int function( const int first, const char second, const float third );

int compute( void )
{
    int  value;

    // Function call sample.
    value = function( 10, 'a', 123.456f );

    // Expression sample.
    return (value + 100) / 2;
}
```

5. Classes

5.1. Préservation de l'encapsulation

Dans le concept de programmation objet, chaque objet est vu comme une boîte noire dont la seule partie visible et accessible est l'ensemble des services proposés. Pour utiliser une telle boîte noire, il n'est pas besoin d'en comprendre le fonctionnement interne, seule l'interface de cet objet est intéressante.

En C++, les services proposé par un objet seront représentés par les fonctions membres de la section publique et de la section protégée.

Règle 35 : Masquer l'implémentation

Mettez le maximum de membres (variables ou fonctions) dans la section privée. Une variable non statique constante se trouvant dans la section publique est souvent un indicateur d'un problème de conception.

Notez qu'en C++, l'implémentation ne peut pas être complètement masquée, du fait que la section privée d'une classe reste visible dans le fichier d'en-tête.

Règle 36 : Accesseurs

Fournissez des accesseurs plutôt que de mettre une variable disponible dans la section publique ou protégée.

Règle 37 : Nom des accesseurs

Pour vos accesseurs, utilisez des noms permettant de comprendre clairement leur rôle. Pour les accesseurs permettant de retrouver (respectivement mettre à jour) une valeur, commencez l'identifiant par « get » (respectivement « set »).

Notez aussi que les accesseurs permettant de retrouver une valeur devront être des fonctions constantes (cf *Mot clef « const »*, page 21).

Règle 38 : Constructeurs

Fournissez des constructeurs de bonne facture, permettant d'initialiser intégralement une instance, par opposition aux objets construits dans un état neutre et nécessitant ensuite l'appel successifs de plusieurs fonctions, parfois dans un ordre précis, avant d'être viables.

Recommandation 18 : Libération des ressources dans le destructeur et les opérateurs d'affectation

Dans un opérateur d'affectation, la première opération consiste à réinitialiser l'objet (libération des ressources) avant de lui affecter le nouvel état. Or c'est aussi le travail du destructeur. Il est donc recommandé de définir une fonction qui centralise les opérations de libération de ressources et qui assiste le destructeur et les opérateurs d'affectation dans cette tâche. Cette fonction sera appelée « destroy ».

Recommandation 19 : Accesseurs même en interne

Au lieu d'utiliser directement les variables membres au sein d'une classe, passez par les accesseurs. Ceci fournit un petit niveau d'abstraction et offre la possibilité de faire quelques vérifications de manière centralisée au moment de l'accès aux données.

Exemple 23 : Accesseurs

```
class Sample
{
public:
    const std::string& getName( void ) const;
    void               setName( const std::string& name );

private:
    std::string   m_name;
};
```

Exemple 24 : Bon constructeur

```
// Declaration
class Sample
{
public:
    Sample( const std::string& name );
    void setName( const std::string& name );
};

...

// Usage
Sample   sample( "name" );
```

Exemple 25 : Destructeur et opérateurs d'affectation

```
class Sample
{
public:
    ~Sample( void );
    void operator=( const Sample& sample );

private:
    void destroy( void );
};


Sample::~Sample( void )
{
    destroy();
}


void Sample::operator=( const Sample& sample )
{
    destroy();

    ...
}
```

5.2. Divers

Règle 39 : Construction d'une instance

Lors de la construction d'une instance de classe, utilisez la forme appelant le constructeur, plutôt que la forme utilisant une affectation. Le même code sera généré pour les deux versions (i.e. un appel à un constructeur) mais à la lecture du code, la forme utilisant l'affectation peut entraîner une confusion avec

une affectation réelle.

Règle 40 : Protection contre la copie

Si vous définissez une classe ayant des variables membres pour lesquelles le constructeur de copie par défaut ne fournit pas le comportement adéquate (comme pour les pointeurs), et que vous voulez pas autoriser la copie d'instance de cette classe, définissez alors un constructeur de copie vide que vous placerez dans la section privée. Ainsi, toute tentative de copie d'objets de cette classe sera interdite à la compilation.

Règle 41 : Initialisation des membres

Vous utiliserez le plus possible les initialiseurs dans les constructeurs de vos classes. Cela vous permettra de réaliser de manière uniforme les appels aux constructeurs des classes de base, ainsi que la construction des objets membres.

Notez que l'ordre dans lequel vous écrivez les initialiseurs ne définit pas l'ordre réel dans lequel ils vont être appelés en réalité. Ce dernier est en effet basé sur l'ordre de déclaration des membres de la classe. Certains compilateurs émettent un avertissement si l'ordre dans la déclaration des membres et dans les listes d'initialiseurs ne correspondent pas.

Recommandation 20 : Pointeur « `this` »

Il n'est pas recommandé d'utiliser « `this->` » pour signifier l'accès à un membre d'une instance à partir de cette instance même.

Bien que cette information puisse être utile, elle peut également devenir une contrainte durant la maintenance d'un programme. Imaginons qu'un membre d'une classe soit déplacé vers une classe de base, tout en restant accessible pour ses classes dérivées, il faudra alors mettre à jour une certaine quantité de code dans différentes classes afin de garantir la consistance de l'utilisation de « `this->` ».

Enfin, cette information n'a pas de sens quand « `this->` » référence une fonction virtuelle pour laquelle on ne peut pas connaître la classe qui fournira l'implémentation réellement exécutée sans connaître le type réel de l'instance.

Recommandation 21 : inline

L'utilisation du mot-clef `inline` est facultative étant donné que la norme du C++ ne définit rien à ce sujet. Chaque compilateur a son interprétation spécifique. Il est à noter que lors des phases d'optimisation, les compilateurs intelligents se chargeront automatiquement de rendre certaines fonctions « inlines » et ceci de façon pertinente.

Exemple 26 : Construction d'une instance

```
// Bad instance construction
Sample   instance = Sample( "name" );

// Good instance construction
Sample   instance( "name" );
```

Exemple 27 : Protection contre la copie

```
class Sample
{
private:

    Sample( const Sample& sample ) {}
};
```

Exemple 28 : Initialisation des membres

```
Sample::Sample( const std::string& name )
:   m_name( name )
{}
```

6. Divers

Cette dernière partie regroupe enfin tout ce qui n'a pas trouvé sa place ailleurs. Ceci étant dit, les points abordés ici n'ont pas pour autant moins d'importance.

Règle 42 : Namespaces

Vous organiserez votre code au sein de namespaces. Par défaut, vous aurez au moins un namespace pour votre module (application ou librairie). Dans ce namespace, vous êtes encouragés à scinder votre code dans des sous-namespaces, et ainsi de suite.

Les namespaces peuvent être vus comme des délimiteurs de modules. Les namespaces permettent de faire cohabiter des types qui ont le même nom mais qui sont issus de modules différents, et qui n'ont par conséquent par nécessairement le même rôle.

Veuillez vous référer à la documentation officielle.

Règle 43 : Mot clef « `const` »

Définissez autant que possible des paramètres constants, des fonctions constantes et des variables membres constantes (cf *Constantes*, page 21).

Il arrive que cette fonctionnalité du C++ ne soit pas utilisée, souvent par simple ignorance de son existence. Son utilisation n'apporte absolument rien au code généré par le compilateur. Par contre, lors de la phase d'analyse du code source, le compilateur sera à même de faire une vérification des accès des algorithmes. Ainsi, une fonction déclarée constante ne sera pas autorisée à modifier l'état d'un objet, un paramètre déclaré constant ne pourra pas être modifié par une fonction.

Règle 44 : Constantes

Utilisez des objets statiques constants ou des énumérations comme constantes, à la place de traditionnelles directives `#define`.

Ceci permettra au compilateur d'effectuer des vérifications de type. Lors de débogages, vous serez à même de visualiser les valeurs des constantes, ce qui vous aidera sans doute dans votre travail. Vous pouvez avoir le contrôle de la portée de vos constantes et les cantonner à un namespace, une classe (avec accès public, protégé ou privé). Enfin, les constantes ne seront pas limitées aux types scalaires seuls, mais également aux types utilisateurs.

Règle 45 : Références vs pointeurs

Quand c'est possible, utilisez des références plutôt que des pointeurs.

Prêtez attention aux cas où ces mêmes références peuvent être constantes (cf *Mot clef « const »*, page 21).

Règle 46 : Conversions de types

Pour les conversion de types, utilisez les opérateurs du C++ fournis à cet effet et qui sont `dynamic_cast`, `static_cast`, `const_cast` et `reinterpret_cast` (l'utilisation de ce dernier étant à éviter).

Recommandation 22 : Ne pas réinventer la roue

Réinventer la roue n'a pas de sens. Parmi tous les outils disponibles (logiciels, librairies, ...) il y en a certainement une qui répond à un de vos besoins. Alors plutôt que de passer du temps pour mettre au point une nouvelle version d'un algorithme, cherchez à réutiliser ce qui existe et qui a déjà été éprouvé.

Recommandation 23 : Exceptions

Les exceptions sont le mécanisme préféré pour la notification d'erreurs du fait qu'il est bien plus puissant que le simple retour de codes d'erreur.

Recommandation 24 : Raccourcis pour les instanciations « templates » de classe.

A la place d'utiliser directement une instanciation « template » de classe, définissez un raccourcis à l'aide d'un `typedef`. Ceci est particulièrement recommandé pour les instanciations « templates » recevant des types compliqués.

Exemple 29 : Constantes

```
namespace a
{


class Sample
{
public:
 enum
 {
 ENUM_CLASS_CONSTANT_ONE,
 ENUM_CLASS_CONSTANT_TWO,
 };

 static const int  publicClassConstant;

private:
 static const int           privateClassConstant;
 static const std::string   privateClassConstantAsString;
};


} // namespace a
```

Exemple 30 : Mot clef « `const` » et les références (par opposition aux pointeurs)

```
class Sample
{
public:
    /**
     * @ brief    Constant function.
     *
     * This function is an accessor that doesn't affect the object state.
     */
    const std::string& getName( void ) const;

    /**
     * @brief Non constant function.
     *
     * This function is an accessor that update the object state.
     */
    void setName( const std::string &name );
};
```

Cet exemple présente deux fonctions membres, l'une étant déclarée constante, l'autre non. Vous remarquerez que la valeur de retour de la première est une instance constante de classe passée par référence, ce qui est également le cas du paramètre de la deuxième fonction. Cette technique permet de passer des objets par référence, plutôt que par copie, tout en garantissant que ces objets ne sont pas modifiables. Ceci évite ainsi la construction d'instances intermédiaires pouvant être coûteuses en temps

et en mémoire.

Exemple 31 : Raccourcis pour les instanciations « templates » de classe.

```
typedef std::map< std::string, unsigned int > MapAlias;
```