Règles d'écriture en C++

Plan Qualité  
Version 2.0

Sommaire

[1. Convention de codage pour les applications C++ et Java 3](#_Toc282002449)

[1.1. Organisation des fichiers 3](#_Toc282002450)

[1.1.1. Instruction Include 3](#_Toc282002451)

[1.1.2. Multiple inclusions de fichiers de définition 3](#_Toc282002452)

[1.2. Conventions de nommage 3](#_Toc282002453)

[1.2.1. Noms des routines 3](#_Toc282002454)

[1.2.2. Noms des Classes 4](#_Toc282002455)

[1.2.3. Noms des variables 5](#_Toc282002456)

[1.3. Documentation du source 5](#_Toc282002457)

[1.3.1. Commentaires des déclarations de variables 5](#_Toc282002458)

[1.3.2. Commentaires des structures algorithmiques 5](#_Toc282002459)

[1.3.3. Commentaires des routines 5](#_Toc282002460)

[1.4. Conventions de programmation 6](#_Toc282002461)

[1.4.1. Initialisation des données membres d'une classe 9](#_Toc282002462)

[1.4.2. Gestion des erreurs 9](#_Toc282002463)

[1.5. Présentation 9](#_Toc282002464)

[1.5.1. Expressions complexes 10](#_Toc282002465)

[1.5.2. Appels de routines avec de nombreux paramètres 10](#_Toc282002466)

[1.5.3. Déclaration des méthodes d'une classe 11](#_Toc282002467)

[1.6. Test et debug 11](#_Toc282002468)

[1.6.1. Invariant de classe 11](#_Toc282002469)

[1.6.2. Assertions 11](#_Toc282002470)

[1.7. Divers 12](#_Toc282002471)

## Organisation des fichiers

Le nom des fichiers peut avoir plus de 8 caractères (majuscules et minuscules) et doit refléter le contenu du fichier. Chaque fichier de définition et d'implémentation de classes ne doit contenir qu'une classe, sauf pour des classes privées ou contenues dans une autre classe. Son nom est le même que celui de la classe.

**Extensions de fichiers standards :**

.c Fichier de définition C

.cpp Fichier de définition C++

.h Fichier d'entête C/C++

.inl Fichier de fonctions inline C++

.idl Langage de description d'interface

.rc Script de ressource

.java Fichier source Java

### Instruction Include

Elle ne doit jamais référencer un chemin absolu.

Erreur : #include "/code/Telephonie/Include/PhoneLine.h"

Correct : **#include "PhoneLine.h"**

Correct : **#include "../Include/PhoneLine.h"**

### Multiple inclusions de fichiers de définition

Les inclusions multiples sont impossibles, il faut donc les rendre conditionnelle en entête du fichier à inclure.

#ifndef \_PhoneLine\_H\_

#define \_PhoneLine\_H\_

... contenu du fichier ...

#endif // \_PhoneLine\_H\_

## Conventions de nommage

### Noms des routines

Suivre les conventions de Java. C'est une concaténation de mots avec la première lettre de chaque mot en majuscule sauf le premier de la chaîne concaténée. Ne pas commencer les noms par "\_" ou "\_\_", réservés respectivement aux librairies et au compilateur.

Routines : ensembleDesMembres(), rapportImpression(), initialisation().

Les fonctions (pas les méthodes) exportées de modules DLL doivent contenir une abréviation en majuscule du nom du module.

**Routines de DLL :** DEBUGTrace(), DBGTrace().

Un nom correct de routine doit décrire clairement ce qu'elle fait.

Les fonctions qui ne retourne qu'un code d'erreur doivent être considérées comme des procédures.

**Pour une procédure** utiliser un verbe fort suivi par un objet.

**Procédures :** imprimerRapport(), calculerRevenusMensuels(), repaginerDocument().

**Pour une méthode**, ne mettre que le verbe.

**Méthodes :** imprimer(), calculer(), repaginer().

**Pour une fonction**, utiliser une description de la valeur retournée.

**Fonction :** cos(), prochainClientID(), ImprimanteLibre(), couleurCouranteStylo().

**Pour une méthode retournant une valeur**, ne pas mettre l'objet.

**Méthodes avec valeur de retour :** cos(), idSuivant(), libre(), couleurCourante().

Eviter les verbes flous : réaliser, traiter, gérer, sauf pour les événements qui peuvent être 'gérés'. Si le nom est difficile à trouver, c'est peut être que le choix structurel de cette routine n'est pas très bon !

Les attributs d'une classe sont accessibles au travers d'**assesseurs 'get' et 'set'** (accessors):

private int solde; // ou m\_solde : donnée membre de la classe

public int getSolde(); // sélecteur

public void setSolde(valeur : int); // modifieur

**Assesseurs :**

Color getCouleurCourante()

void setCouleurCourante(Color c)

bool getPret()

void setPret(bool dispo)

### Noms des Classes

Suivre les conventions de Java. C'est une concaténation de mots avec la première lettre de chaque mot en majuscule. Par exemple: Voiture, Imprimante, NouveauClient.

Les noms de classes accessibles par DCOM seront précédées de la lettre "I".

**Classes accessibles par DCOM :** IWindowModel, IUnknown.

### Noms des variables

Suivre les conventions de Java. C'est une concaténation de mots avec la première lettre de chaque mot en majuscule sauf le premier de la chaîne concaténée. Il doit représenter clairement le contenu de la variable.

**Variables :** nbCaisses, index, vitesseMax.

Les variables qui sont membres d'une classe peuvent être précédées du préfixe **"m\_"** (leurs assesseurs éliminent ce préfixe).

**Exemple :** m\_nbCaisses, getNbCaisses(), setNbCaisses( 10 ), m\_index, m\_vitesseMax.

Les constantes sont une exception à la règle. Ils s'écrivent en majuscules avec '\_' entre les mots.

**Constantes :** LONGUEUR\_MAX, TAILLE\_DU\_NOM, SOLDE\_MINI.

## Documentation du source

/\*

\* Nom de classe : MaClasse

\* Description : description de la classe et de son rôle

\* Version : 1.0

\* Date : 23/02/2001

\* Auteur : moi

\*

\* Révision : Modification de .......

\* Description : bla bla bla

\* Date : jj/mm/aaaa

\* Auteur : un autre

\*/

Il doit y avoir un commentaire en tête de chaque source et au début de chaque routine.

### Commentaires des déclarations de variables

Tout ce que le nom ne suffit pas à décrire : unité, l'intervalle de définition, l'intervalle des valeurs possibles, description de champs de bits, etc.

Pour les données globales faire une documentation très précise.

### Commentaires des structures algorithmiques

Expliquer avant la structure le rôle de cette structure et de son contenu.

### Commentaires des routines

Mettre en entête de chaque routine un commentaire la décrivant et décrivant l'utilisation de données globales si il en a.

Des commentaires commençant par "//" peuvent être ajoutés dans le source pour donner des précisions. Documenter variables locales et paramètres à côté de leur déclaration.

/\*------------------------------------------------------------------------------------------------------------------

\* ....

\* Description : Retourne une valeur aléatoire entière comprise entre deux valeurs

\* données positives ou nulles. Utilise <sys/timeb.h>.

\*

\* PréCdt : min et max sont des entiers positifs ou nuls, max >= min.

\* ------------------------------------------------------------------------------------------------------------------\*/

int random ( // retourne une valeur aléatoire

int min, // entre un minimum inclus

int max // et un maximum inclu

)

struct \_timeb t1; // mémorise l'heure de l'horloge du PC ds une structure

\_ftime( &t1 ); // lecture de l'heure de l'horloge du PC

srand( t1.millitm ); // init de la fonction rand() avec les ms de l'heure

return ( rand()%( max + 1 – min ) + min ); // rand() retourne un entier qcq >=0

**}**

## Conventions de programmation

Limiter l'utilisation des macros au minimum, utiliser plutôt une fonction inline :

inline int square( int x )

{

return ( x \* x );

};

Utiliser des constantes et des types énumérés plutôt que des "#define".

Eviter les données globales.

Une classe qui utilise un "new" afin d'allouer une instance gérée par cette classe doit définir un constructeur par copie.

Utiliser un passage par référence constante (const &) plutôt qu'un passage par valeur.

Une méthode publique qui retourne une valeur par référence doit le faire par const &.

Par défaut, déclarer les rôles des associations **protected**. Ajouter l'assesseur get en cas de besoin.

Ne pas utiliser les fonctions de type C, comme printf(), mais trouver leur équivalent mieux protégée en C++.

Pour les méthodes qui ne retourne rien mettre explicitement **void** (sauf pour le constructeur et le destructeur).

Limiter le nombre d'objets temporaires, car les pointeurs sur ces objets sont difficiles à manipuler.

Eviter des pointeurs sur d'autres pointeurs.

Utiliser **typedef** pour simplifier les déclarations complexes et répétées.

**Différents mécanismes de passage de paramètre**

// a. Une copie du paramètre est réalisée sur la pile, le constructeur de copie est

// appelé en entrée,et le destructeur est appelé en sortie. Peu efficace.

void f1( string s );

string a;

f1( a ); // appel par valeur

// b. Seule la référence est copiée mais f2 peut modifier le paramètre.

void f2( string& s );

string b;

f2( b ); // appel par référence

// c. Seule la référence est copiée et f3 ne peut pas modifier le paramètre.

void f3( const string& s );

string c;

f3( c ); // appel par référence constante, la meilleure solution.

// d. Seule l'adresse est copiée et f4 ne peut pas modifier le paramètre.

// mais dans f4, la syntaxe d'utilisation de 's' risque d'être compliquée.

void f4( const string\* s );

string d;

f4( &d ); // appel par pointeur constant

Limiter les conversions explicites (casts).

Utiliser uniquement les nouveaux opérateurs de transtypage du C++ :

**static\_cast conversion de type avec vérification à la compilation**

**dynamic\_cast conversion polymorphe à l'exécution**

**const\_cast annule la constance des objets**

**reinterpret\_cast transtypages non standards.**

double b = 28.2;

int x = static\_cast< int > ( d ); // x <- 8

La norme du C++ offre une nouvelle écriture des opérateurs :

**&& and**

**|| or**

**! not**

**!= not\_eq**

**& bitand**

**| bitor**

**^ xor**

**~ compl**

**&= and\_eq**

**|= or\_eq**

**^ xor\_eq**

**nécessite sous VC++ l'inclusion de <iso646.h>**

Ne pas faire **if (valeur)** mais **if (valeur != 0)**, même chose pour les pointeurs.

Ne pas faire **if (libre == true)** mais **if (libre)**.

La variable d'une boucle **for** doit avoir la même visibilité que celle ci.

for ( int index = 0; index<10; index++ ) {

cout << index;

}

Dans une structure **switch :** mettre un **break** après chaque **case** et un **default**.

Pour les boucles infinies utiliser **for ( ; ; )**.

Si vous créez une librairie de fonction, les mettre dans une classe utilitaire (ces membres sont tous **static**).

Si une valeur numérique est utilisée au moins deux fois, déclarer une constante.

Pour des variables d'état utiliser le type **bool** et les valeurs **true** et **false**.

Utiliser **typeid** pour connaître le type d'une variable ou d'une classe à l'exécution.

Le manipulateur de flux **boolalpha** affiche correctement la valeur d'un booléen :

bool pret = true;

cout << boolalpha << pret << endl; // affiche true

Par défaut, on utilisera les types de données de base du C++ pour les déclarations :

**long double réel 80 bits (spécifique VC++)**

**double réel 64 bits**

**float réel 32 bits**

**unsigned long entier non signé 32 bits**

**long entier 32 bits**

**unsigned entier non signé 16 ou 32 bits**

**int entier 16 ou 32 bits**

**unsigned short entier non signé 16 bits**

**short entier 16 bits**

**unsigned char entier non signé 8 bits**

**char caractère (entier 8 bits)**

**bool booléen (true ou false)**

Attention, toute conversion d'une valeur d'un type vers un type plus 'petit' peut entraîner une erreur.

Ex : La catastrophe d'Ariane 5. Une conversion d'un entier 64 bits en entier 16 bits d'une valeur supérieure à 2^15, représentant "le décalage horizontal" (important sur Ariane 5), dans le système de référence inertielle écrit en ADA a provoqué une exception non interceptée et boom !

Le commentaire sur le danger de cette conversion était caché au cœur du source et non dans l'entête du module (le relecteur ne l'a pas vu et tous les tests de validation ont fonctionnés).

### Initialisation des données membres d'une classe

Utiliser l'initialisation du constructeur, plutôt que l'affectation. C'est d'ailleurs la seule façon d'affecter des références.

MaClass::MaClass() : nbTab( 3 ), couleurColor( red ), couleur( red ) {

// ne pas faire nbTab = 3;

}

### Gestion des erreurs

Une classe ne doit pas être laissée dans un état indéfini (un objet fichier qui n'a pas pu ouvrir le fichier disque associé, par exemple).

Les erreurs affectant le traitement de la routine doivent être gérées et le client doit en être informé si cela a des conséquences pour lui.

Le paramètre contenant le code de l'erreur doit être distinct de tous les autres paramètres (pas de valeur 0 d'un pointeur utile, etc.).

## Présentation

Formater vos sources correctement et tous de la même façon !

Ne mettre qu'une seule instruction par ligne.

Toute structure alternative ou itérative est suivie par un bloc "{ …}", chaque accolade est alignée sous le début de la structure (même si il n'y a qu'une instruction ou aucune).

Coller **\*** et **&** au type pour une déclaration et au pointeur pour un traitement.

Pointeurs et référence : int\* ptr, string& texte, \*ptr = 12

Mettre les tabulations à 4 espaces et configurer l'éditeur pour qu'il remplace les tabulations par des espaces (sous VC++ faire : View > Properties > Tab size: 4 et Indent size : 4).

void verifier( int parametre1, string parametre2 )

{

char currentChar;

faireQuelqueChose();

while (condition) {

faireQuelqueChose();

faireAutreChose();

if (condition) {

faireQuelqueChose();

switch (condition) {

case CAS\_1:

faireQuelqueChose();

break;

case CAS\_2:

faireQuelqueChose();

break;

default:

faireQuelqueChose();

} // switch

} // while

}

### Expressions complexes

Pour des conditions complexes, les mettre à raison d'une par ligne.

if ( ( ?0' <= car && car <= ?9' ) || ( ?a' <= car && car <= ?z' ) || ( ?A' <= car

&& car <= ?Z' ) ) {

faireQuelqueChose( car );

}

**Doit être remplacé par :**

if ( ( ?0' <= car && car <= ?9' ) ||

( ?a' <= car && car <= ?z' ) ||

( ?A' <= car && car <= ?Z' ) ) {

faireQuelqueChose ( car );

}

### Appels de routines avec de nombreux paramètres

Aligner les paramètres, à raison de un par ligne :

dessineUneLigne( Window.north,

Window.south,

Window.east,

Window.west,

currentWidth,

currentHeight );

ou

laFenetreClient.dessineUneLigne (

Window.north,

Window.south,

Window.east,

Window.west,

currentWidth,

currentHeight,

normalBackgroundFillColor );

Ne mettre qu'une instruction par ligne.

### Déclaration des méthodes d'une classe

Les déclarations sont faites dans l'ordre **public, protected, private**.

## Test et debug

### Invariant de classe

C'est une expression booléenne qui doit toujours être vrai.

Chaque classe devrai fournir une fonction booléenne pour vérifier l'invariant (l'état normal) de celle-ci.

### Assertions

Toute routine non triviale devrait contenir les appels suivant avant tout traitement :

Test d'invariance,

Une assertion de pré condition contenant un test de validité des paramètre et des conditions initiales nécessaire au traitement demandé.

Ces tests utilisent les macros **\_ASSERT(), ASSERT()** ou la fonction **assert()** (resp. VC++, MFC, ANSI). A ne pas utiliser pour traiter autres choses (les erreurs par exemple).

On peut utiliser **assert( false )** pour s'assurer qu'une section de code n'est jamais exécutée (méthode virtuelle qui doit être surchargée, default dans un switch()).

Dans certains cas, on peut utiliser **assert()** pour des problèmes de capacités mémoire :

ptr = new int[ taille ];

assert ( ptr != 0 ); // on arrête tout !

Attention les assertions ne sont plus traitées à l'exécution finale, ne pas y mettre de traitement autre que le test.

## Divers

Utiliser la librairie standard **STL** lorsque le service recherché s'y trouve (string, collection, ..).

Utiliser le nouvel espace de nom **std** et les fichiers de définition associés (**<string>** sans ".h").

Références : <http://www.macadamian.com/codingconventions.htm> et d'autres.