POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

GUIDE DE CODAGE POUR *C++*

18 MARS 2024

CHARLES DE LAFONTAINE, JÉRÔME COLLIN, MERIAM BEN RABIA, LOUIS GRANGER, MARTIN BISSON, NOUREDDINE KERZAZI, MICHEL GAGNON
GIGL | Polytechnique Montréal

TABLE DES MATIÈRES

GUIDE DE CODAGE POUR *C++*

1. INTRODUCTION

Ce document énumère les recommandations de codage relatives au langage *C++* communément admises dans la communauté de développement. Pour tout commentaire ou suggestion, vous pouvez communiquer avec Jérôme Collin.

Au niveau de l'importance des normes et des recommandations, les directives qui se retrouvent dans ce document n'ont pas toutes la même importance. Certaines doivent être respectées, alors que d'autres sont plutôt souhaitables.

1.1. RAISONS DU DOCUMENT

Il y a plusieurs raisons qui font la nécessité de normaliser la façon de rédiger du code, entre autres :

1. Augmenter la lisibilité et la compréhension du code source;
2. Avoir un code prédictible et facilement modifiable.

1.2. DISPOSITION DES NORMES ET RECOMMANDATIONS

Les recommandations sont groupées par matière et toute norme ou recommandation est numérotée pour lui faciliter l'accès pendant les mises à jour. La disposition pour les normes et recommandations est comme suit :

 # Brève description de la norme.

```
Un exemple de code, s'il y a lieu…
```

Motivations, contexte et informations additionnelles…

 # Brève description de la recommandation.

```
Un exemple de code, s'il y a lieu…
```

Motivations, contexte et informations additionnelles…

1.3. RÈGLE GÉNÉRALE

1 On permet n'importe quelle violation du guide si elle augmente la lisibilité.

Le but principal de la recommandation est d'améliorer la lisibilité, l'organisation et la qualité générale du code. Il est impossible de couvrir tous les cas spécifiques d'un guide général. Le programmeur se doit d'être **flexible**.

Le lecteur est invité à **faire preuve de jugement**. Si une pratique est inutile, inapplicable, voire contre-productive au regard des contraintes d'un projet ou des objectifs, soyons assez intelligents pour y déroger.

2. CONVENTIONS DE NOMENCLATURE

Cette section énumère les règles à suivre pour assurer la lisibilité et la cohérence pour tous les noms que vous donnerez à vos variables, fonctions, classes, etc.

2.1. CONVENTIONS GÉNÉRALES DE NOMENCLATURE

2 **Le nom d'une classe doit respecter le « PascalCase ».**

```
Ligne, SystemeAudio, PointDeControle
```

3 **Le nom d'une variable doit respecter le « camelCase ».**

```
ligne, application, compteur, compteurDeLigne

Point point;
```

4 **Le nom d'une constante globale doit respecter le « SCREAMING_SNAKE_CASE ». Le nom d'une constante locale doit respecter le « camelCase ».**

```
// Constantes globales
VITESSE_MAX, COULEUR_ROUGE, COULEUR_BLEU, PI
// Constantes locales
vitesseMax, couleurRouge, couleurBleu, pi
```

5 **Le nom d'une méthode doit respecter le « camelCase » en commençant par un verbe.**

```
getName()
obtenirInstance()
lireNom()
calculerLargeurTotale()
```

6 **Le nom d'un espace de nommage (« `namespace` ») doit être en minuscules.**

```
model::analyzer
io::iomanager
common::math::geometry
```

7 **Un type générique doit être représenté par une simple lettre majuscule.**

```
template <typename T>
template <typename C, typename D>
```

8 On devrait utiliser le mot « `typename` » plutôt que « `class` » dans la déclaration d'un type générique.

```
template <typename T> // À ÉVITER : template <class T>
```

Comme le type générique peut être instancié autant par une classe que par un type primitif, on évite ainsi une confusion.

9 Une abréviation doit être en minuscules quand elle est utilisée comme nom de variable.

```
exportHtmlSource(); // À ÉVITER : exportHTMLSource();
openDvdPlayer();    // À ÉVITER : openDVDPlayer();
ouvrirLecteurDvd(); // À ÉVITER : ouvrirLecteurDVD();
```

L'utilisation des majuscules pour les noms propres **viole la règle précédente et diminue la lisibilité du code source**. Un autre problème : lorsque le nom est concaténé à un autre, la lisibilité est énormément réduite si on met l'abréviation entièrement en majuscules, car le nom qui suit l'abréviation ne ressort pas comme il le devrait.

10 On doit toujours se référer à une variable globale en utilisant l'opérateur de portée « :: ».

```
::fenetrePrincipale.ouvrir();

::uneVariableGlobale = 8;
```

Malgré cette règle, en général, **l'utilisation des variables globales devrait être évitée**. On doit plutôt utiliser des constantes pour éviter un ajout en espace de nommage.

11 Le nom d'un attribut privé d'une classe doit débuter ou se terminer par le caractère souligné « _ ».

```
class UneClasse
{
      private : int longueur_;
}
```

Le fait d'indiquer une portée de classe en utilisant le caractère souligné facilite la distinction entre les variables membres de la classe et les variables locales. Cela est important, car les membres d'une classe ont généralement une signification plus importante que les variables locales et doivent donc être traitées avec plus d'attention par le programmeur. Un effet secondaire de cette convention est qu'elle règle élégamment le problème de trouver un nom raisonnable pour les noms des paramètres des constructeurs et des méthodes servant à modifier la valeur d'un attribut :

```
void setDepth(int depth) { // Style à préférer (nous lisons de
      depth_ = depth;         // gauche à droite)
}

void setDepth(int depth) { // Style aussi utilisé par certains
      _depth = depth;         // développeurs [lecture de 1ère vue réduite]
}
```

On se demande parfois si le caractère souligné devrait être ajouté comme préfixe ou comme suffixe. **Les deux pratiques sont utilisées couramment, mais le suffixe est recommandé, car il préserve davantage la lisibilité du nom. Dans tous les cas, il ne devrait jamais y avoir de mélange de nomenclature** (à la fois le caractère « _ » au début **et** à la fin pour au moins deux noms d'attributs) **au sein d'un projet.**

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

```
void setTopic(Topic topic)
/*
  À ÉVITER :
  void setTopic(Topic value)
  void setTopic(Topic aTopic)
  void setTopic(Topic t)
*/
```

```
void connect(Database database)
/*
  À ÉVITER :
  void connect(Database db)
  void connect(Database oracleDb)
*/
```

Réduit la complexité par la réduction des différents noms utilisés. Aussi, on déduit facilement le type de la variable, à la simple lecture de son nom. Si vous ne pouvez respecter cette directive, vous avez probablement mal choisi le nom de votre type. Les variables non génériques jouent un rôle. Ces variables peuvent être nommées en combinant le rôle et le type.

```
Point  pointDepart, pointCentre;
Nom    nomUsager;
```

13 Les noms doivent être tous en anglais ou tous en français.

Même si l'anglais est la langue universelle des développeurs, la langue française reste un bon choix (dépendamment du **projet** et du **client**). Cependant, **nous ne recommandons pas les programmes bilingues**.

14 Une variable de longue portée devrait avoir un nom qui est long. Celle de portée réduite devrait avoir un nom qui est court.

Les variables employées pour un stockage de données temporaires ou pour l'utilisation des indices devraient avoir un nom court. Un programmeur lisant une telle variable devrait être capable de supposer que sa valeur n'est pas employée au-delà de ces lignes de code. En général, les variables temporaires pour des entiers sont « `i` », « `j` », « `k` », « `m` », « `n` », et pour des caractères, « `c` » et « `d` » (normalement au sein de boucles itératives).

15 Le nom de classe d'un objet est implicite et ne doit pas figurer dans le nom d'une méthode.

```
line.getLength();     // À ÉVITER : line.getLineLength();
ligne.lireLongueur(); // À ÉVITER : ligne.lireLongueurLigne();
```

La présence du nom de la classe devient **redondante** avec le nom de l'objet.

2.2. NOMENCLATURE SPÉCIFIQUE

16 Les termes {« obtenir », « modifier »} ou {« lire », « écrire »} doivent être employés là où un attribut est accessible directement.

```
employe.obtenirNom();
employe.modifierNom(nom);
matrice.obtenirElement(2, 4);
matrice.modifierElement(2, 4, valeur);
```

L'équivalent anglophone est « *set* » et « *get* ».

17 Le terme « calculer » (« *compute* ») doit être employé au sein du nom d'une méthode où un calcul est effectué.

```
ensembleValeurs.calculerMoyenne();
matrix.computeInverse();
```

Cela donne au lecteur un indice immédiat que cette opération consomme potentiellement du temps. Si ladite opération est employée souvent, le lecteur pourrait songer à utiliser la cache du résultat. L'emploi conséquent du terme améliore la lisibilité.

18 Le terme « trouver » (« *find* ») doit être employé au sein du nom d'une méthode où une recherche est effectuée.

```
sommet.trouverSommetVoisin();
matrix.findSmallestElement();
noeud.trouverCourtChemin(Noeud noeudDestination);
```

Cela fournit au lecteur un indice immédiat qu'il s'agit d'une méthode de recherche avec un minimum de calcul. L'emploi conséquent du terme améliore la lisibilité.

19 Le terme « initialiser » (« *initialize* ») doit être employé au sein du nom d'une méthode servant à initialiser l'état d'un objet.

```
imprimante.initialiserStyle();
```

L'abréviation « *init* » devrait être évitée.

20 Le nom d'une variable représentant une composante de l'interface usager (GUI) doit être suffixé par le type de l'élément en anglais. En français, il doit être préfixé.

```
mainWindow, propertiesDialog, widthScale, loginText, leftScrollbar,
mainForm, fileMenu, minLabel, exitButton, yesToggle, // …

fenetrePrincipale, dialoguePropriete, echelleLargeur, texteLogin,
barreDefilementGauche, formulairePrincipal, menuFichier, labelMin,
boutonSortie, toucheBasculeOui, // …
```

Cela améliore la lisibilité. À partir du nom, l'utilisateur a une indication immédiate sur le type de la variable et les ressources disponibles de l'objet.

21 Toute collection doit avoir un nom au pluriel.

```
vector<int> points;
int         valeurs[];
```

Cela améliore la lisibilité. À partir du nom, l'utilisateur a une indication immédiate sur le type de la variable et les opérations qui peuvent être exécutées sur les éléments internes à ladite collection (conteneur, tableau).

22 Le préfixe « n » doit être employé au sein du nom d'une variable représentant un nombre d'objets.

```
nPoints, nLignes
```

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

23 Le préfixe « no[s] » (ou le suffixe « *No*[*s*] » en anglais) doit être utilisé au sein du nom d'une variable représentant un numéro d'entité.

```
EmployeeNo, noEmploye
```

La notation est empruntée des mathématiques où il existe une convention établie pour indiquer un nombre d'entité.

24 Une variable itérative doit être appelée « i », « j », « k », etc.

```
for (int i = 0; i < nTables); i++) {
      // …
}

for (vector::iterator i = list.begin(); i != list.end(); i++){
      Element element = *i;
      // …
}
```

La notation est empruntée des mathématiques où il existe une convention établie pour indiquer les itérateurs. Les variables nommées « `i` », « `j` », « `k` », etc., devraient être employées pour des boucles imbriquées seulement.

25 Le préfixe « est » (« *is* ») doit être employé au sein du nom d'une variable ou méthode booléenne.

```
estVisible, estActif(), estTrouve, estOuvert()
   isVisible, isActive(), isFound, isOpen()
```

L'utilisation du préfixe « est » résout un problème commun du mauvais choix du nom de la variable booléenne comme « `ÉTAT` » ou « `INDICATEUR` ». L'utilisation de « `estEtat` » ou « `estIndicateur` » simplement n'est pas suffisante, le programmeur est forcé de choisir des noms plus significatifs. L'équivalent anglophone est « *is* ». Il existe des alternatives au préfixe « est » qui sont appropriées dans certaines situations. Ce sont les préfixes « a » (« *has* »), « peut » (« *can* ») et « doit » (« *must* ») :

```
bool aLicence();
bool peutEvaluer();
bool doitQuitter = false;
```

26 Le nom d'une méthode et de variable devrait favoriser la symétrie.

- En **anglais** :
  `get`/`set`, `add`/`remove`, `create`/`destroy`, `start`/`stop`, `insert`/`delete`, `increment`/`decrement`, `old`/`new`, `begin`/`end`, `first`/`last`, `up`/`down`, `min`/`max`, `next`/`previous`, `open`/`close`, `show`/`hide`, `suspend`/`resume`, etc.
- En **français** :
  `obtenir`/`modifier`, `ajouter`/`retirer`, `creer`/`detruire`, `demarrer`/`arreter`, `incrementer`/`decrementer`, `ancien`/`nouveau`, `debut`/`fin`, `premier`/`dernier`, `haut`/`bas`, `min`/`max`, `prochain`/`precedent`, `ouvrir`/`fermer`, etc.

Cela réduit la complexité par la symétrie des noms de ces entités.

27 Toute abréviation doit être évitée dans le nom.

```
calculerMoyenne();  // À ÉVITER : calcMoy();
```

Il y a deux types de mots à considérer. Tout d'abord, les mots relativement communs, généralement énumérés dans un dictionnaire non technique, ne doivent jamais être abréviés. Évitez d'écrire :

- `cmd` à la place de `commande`
- `calc` à la place de `calculer`
- `cp` à la place de `copie`
- `e` à la place d'`exception`
- `init` à la place d'`initialiser`
- `pt` à la place de `point`
- etc.

De plus, certains termes, parfois spécifiques à un domaine particulier, sont davantage connus sous leur forme abrégée ou par leur acronyme. On devrait garder ces derniers sous leur forme courte. Ne jamais écrire :

- `HypertextMarkupLanguage` à la place de `HTML`
- `CentralProcessingUnit` à la place de `CPU`
- `ProduitInterieurBrut` à la place de `PIB`
- etc.

28 On doit éviter d'ajouter un préfixe (ou suffixe) comme « p » ou « ptr » à un pointeur.

```
Ligne* ligne;
Ligne* pLigne;    // À ÉVITER!
Ligne* lignePtr;  // À ÉVITER!
```

Une convention de nomenclature spécifique aux pointeurs est presque impossible à suivre, car de nombreuses variables sont des pointeurs dans un environnement *C*/*C*++. De plus, les objets en *C*++ sont souvent des types opaques dont l'implantation spécifique devrait être inconnue du programmeur. Le nom devrait dénoter le type seulement lorsque ce dernier a une signification spéciale.

29 La négation au sein du nom de variable booléenne doit être évitée.

```
bool estNonErreur; // À ÉVITER!
bool estNonTrouve; // À ÉVITER!
```

Le problème survient quand on utilise l'opérateur de négation. Il en résulte une double négation plus difficile à comprendre, par exemple « `!estNonErreur` ». Il est beaucoup plus clair d'utiliser une variable booléenne comme « `estErreur` » et d'utiliser l'expression « `!estErreur` » au lieu d'une variable booléenne « `estNonErreur` ».

30 Le nom de classe énumérative ne doit pas être répété au sein du nom d'un état. Le nom d'un état doit être déclaratif.

```
// Bien
enum class Color  { RED,      GREEN,      BLUE };
enum class Couleur      { ROUGE,    VERT,       BLEU };

// À éviter
enum class Color { COLOR_RED, COLOR_GREEN, COLOR_BLUE };
```

Cela évite tout pléonasme, tout en augmentant la lisibilité et la compréhensibilité. L'accès en devient trivial :

```
Color::RED;
Couleur::ROUGE;
```

31 Une classe couvrant une exception doit se terminer par le suffixe « *Exception* » (en anglais) ou débuter par le préfixe « Exception » (en français).

```
class AccessException { … };
class ExceptionAcces    { … };
```

Les classes d'exception ne font pas réellement partie du modèle principal du programme. Le fait de les nommer ainsi les isole des autres classes.

32 Toute méthode retournant un objet doit être nommée d'après ce qu'elle retourne, alors qu'une procédure (méthode ne retournant rien, c'est-à-dire « `void` »), d'après ce qu'elle fait.

Cela améliore la lisibilité. De plus, cela permet de clarifier ce que la méthode devrait faire et également ce qu'elle n'est pas censée faire, permettant d'éviter plus facilement les effets secondaires non souhaités.

2.3. LES FICHIERS

33 Un fichier d'en-tête *C++* devrait avoir l'extension « `.h` » ou « `.hpp` ». Un fichier source devrait avoir l'extension « `.cpp` » ou « `.c++` ».

`MaClasse.cpp, MaClasse.hpp`

Ce sont les extensions généralement privilégiées en *C++*. Évidemment, il en existe d'autres, telles que « `.cc` », « `.cp` », « `.cxx` », « `.hh` », etc. L'important est d'uniformiser vos extensions pour la portée d'un projet.

34 Une classe devrait être déclarée dans un fichier d'en-tête et définie dans un fichier source. Le nom d'un fichier devrait correspondre au nom de la classe.

```
MaClasse.h, MaClasse.c++
```

Facilite le repérage des fichiers associés à une classe donnée. **Une exception évidente à cette règle sont les classes génériques (« *template* ») qui peuvent être déclarées et définies dans un fichier `.h` (d'où la recommandation).**

La convention de nommage, que ce soit « *PascalCase* », « *kebab-case* », « *camelCase* » ou « *snake_case* », demeure flexible, malgré qu'on privilégie généralement le « *PascalCase* » à la fois pour vos noms de fichiers et de dossiers, encore dans le cadre de projets *C++*. Peu importe votre convention sélectionnée, assurez-vous néanmoins que tous vos fichiers y adhèrent dans le cadre d'un seul et même projet.

35 Toute définition doit se trouver dans un fichier source (« .cpp »).

```
class MaClasse // Au sein d'un .h…
{
public :
      int getValue() { return value_; }  // À ÉVITER!
      // …
private:
      int value_;
}
```

Les fichiers d'en-tête doivent déclarer une interface que les fichiers sources doivent l'implanter. Pour les fonctions « *inline* », on peut déroger de cette directive puisqu'elles peuvent apparaître dans le « .h ».

36 Il faudrait éviter qu'une ligne de code soit trop longue (on essaie de ne pas dépasser 80 colonnes).

Ce nombre de colonnes est très commun pour les différents éditeurs, émulateurs de terminal, imprimantes et débogueurs. Ainsi, les fichiers partagés entre différents développeurs devraient respecter cette contrainte. Cela évite la perte de lisibilité qui peut se produire lorsque des retours de chariot non intentionnels se produisent sur les lignes trop longues quand un fichier passe d'un programmeur à l'autre.

Il s'agit d'une recommandation, étant donné qu'en programmation web, pour des fichiers *JavaScript* et *TypeScript*, la norme tend **parfois** à s'étendre à **120** colonnes (*qui devrait être d'ailleurs le plafond maximal de tout fichier nonobstant la langue*). Cela dit, pour un projet, veuillez vous en tenir à une norme commune de nombre de colonnes maximales, et ce, pour tous vos fichiers.

37 Les caractères spéciaux comme « `TAB` » et le saut de page devraient être évités.

Ces caractères risquent de causer des inconsistances entre les éditeurs, imprimantes, émulateurs de terminal ou débogueurs s'ils sont utilisés dans un environnement multi-plateformes.

38 La césure d'une ligne trop longue doit être effectuée d'une manière lisible, logique et évidente.

```
somme = uneVariableDontLeNomEstTresLong +
        uneDeuxiemeVariableDontLeNomEstTresLong +
        uneDerniereVariableDontLeNomEstTresLong;

unObjetQuelconque.uneMethode(unPremierParametre,
                             unAutreParametre,
                            unDernierParametre);

unObjetQuelconque.uneMethodeDontLeNomEstPlutotLong(param1,
                                                   param2,
                                                   param3);

setText("Ligne très très très très très très très longue " +
        "coupée en deux parties.");

if (unObjetQuelconque.obtenirUnDeSesComposants().estActif() ||
    unAutreObjet.obtenirUnDeSesComposants().estActif()) {
      // …
}
```

Les lignes coupées se produisent lorsqu'un énoncé dépasse la limite tolérée pour la longueur d'une ligne. Il est difficile de donner des règles strictes sur la manière de couper les lignes, mais les exemples ci-dessus reflètent l'idée générale.

L'important est de se rappeler qu'un des buts principaux est d'augmenter la lisibilité :

- couper après une virgule;
- couper après un opérateur;
- aligner la nouvelle ligne avec le début de l'expression de la ligne précédente.

2.4. FICHIERS D'EN-TÊTE ET ÉNONCÉS D'INCLUSION

39 Un fichier d'en-tête doit contenir une garde d'inclusion multiple.

```
#ifndef NOMCLASSE_H
      #define NOMCLASSE_H
      // …
#endif // NOMCLASSE_H
```

Permet d'éviter les erreurs de compilation. La convention pour le nom à définir reflète l'endroit où se trouve le fichier dans l'arbre des fichier sources et prévient ainsi les conflits de noms. Ainsi, nous évitons d'écrire « `#pragma once` », qui est **agnostique**, affectant tous les fichiers d'en-tête à être inclus qu'une seule fois; **les redéfinitions multiples fautives deviendraient absorbées**.

40 Les énoncés d'inclusion devraient être ordonnés (par leur position hiérarchique dans le système, avec les fichiers de bas niveau inclus en premier) et groupés. On laisse une ligne vide entre les groupes d'énoncés.

```
#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 montrer au lecteur les fichiers d'inclusion, cette norme donne une indication immédiate à propos des modules impliqués. **Les chemins indiquant l'endroit où se trouvent les fichiers d'inclusion ne doivent jamais être absolus.** On doit plutôt utiliser des directives spécifiques au compilateur utilisé pour indiquer les répertoires de base des inclusions.

41 Un énoncé d'inclusion doit se trouver seulement au début d'un fichier.

Cela évite les effets non désirés causés par des énoncés d'inclusions « cachés » profondément dans un fichier source.

2.5. LES DÉCLARATIONS

2.5.1. Les types

42 Les membres d'une classe doivent être ordonnés de la manière suivante : « `public` », « `protected` » et « `private` ». Chaque section doit être identifiée explicitement. Les sections non applicables ne doivent pas être mentionnées.

En général, lorsqu'on utilise une classe qui a été développée par d'autres, on a un point de vue d'utilisateur. Ce qui nous intéresse est l'ensemble des méthodes que l'on peut utiliser avec les objets de cette classe. D'où l'intérêt de fournir en premier les éléments publics.

Un peu moins fréquemment, on est appelé à définir des sous-classes pour étendre les fonctionnalités d'une classe déjà fournie, ce qui requiert alors un accès aux éléments protégés.

Finalement, comme les éléments privés ne concernent que les développeurs de la classe, ce qui représente généralement un nombre plus restreint de personnes, il est logique de les mettre en dernier.

43 La conversion de type doit toujours être faite de façon explicite. On ne doit jamais dépendre de la conversion implicite.

```
floatValeur = static_cast<float>(intValeur);
// À ÉVITER : floatValeur = intValeur;
```

Ainsi, le programmeur indique qu'il est conscient de la différence entre les types impliqués et que l'utilisation mixte est **intentionnelle**.

2.5.2. Les variables

44 Une variable devrait être initialisée lorsqu'elle est déclarée.

```
int   x           = 0;        // À ÉVITER : int x;
void* buffer      = nullptr; // À ÉVITER : void* buffer;
```

Parfois, il est impossible d'initialiser une variable à sa déclaration. Dans ces cas, la variable devrait être laissée non initialisée plutôt que de l'initialiser à une valeur qui n'a pas de signification.

45 L'utilisation de variables globales doit être évitée.

Une exception pourrait être appliquée aux variables volatiles pour certains systèmes embarqués. Si ces dernières sont absolument nécessaires à l'avènement du projet, elles devraient néanmoins être regroupées au sein d'un espace de nom (« *namespace* »).

46 L'utilisation de fonctions globales doit être minimisée.

En *C++*, il est toujours préférable d'utiliser des méthodes de classes. Ce n'est que dans certains cas exceptionnels, comme pour la surcharge d'opérateurs, qu'on ne pourra pas éviter l'utilisation de fonctions globales.

47 Une variable membre d'une classe ne doit jamais être déclarée publique.

Le concept d'encapsulation est brisé par les variables publiques. Il est préférable d'utiliser des variables privées et des fonctions d'accès.

Une classe qui est essentiellement une simple structure de données, sans comportement (l'équivalent d'une « `struct` » en *C*, **d'interfaces**), pourrait être une exception à cette règle. Dans ce cas particulier, il est approprié de mettre publiques les variables membres de la classe.

48 Les pointeurs et références *C++* devraient avoir leur symbole près du type plutôt que du nom.

```
float*      x = 0.0;    // À ÉVITER : float *x = 0.0;
int&        y = 0;      // À ÉVITER : int &y = 0;
```

La qualité de pointeur ou de référence d'une variable est une propriété du type plutôt que du nom. Les programmeurs *C* utilisent souvent l'approche alternative, tandis qu'en *C++*, il est plus courant de suivre cette recommandation.

49 Le test implicite de comparaison avec « 0 » devrait être utilisé.

```
if (nLignes != 0)      // À ÉVITER : if (nLignes)
if (valeur != 0.0)     // À ÉVITER : if (valeur)
```

Le standard *C++* ne spécifie pas que la valeur nulle pour les types « `int` » et « `float` » corresponde à un « `0` » binaire. De plus, l'utilisation du test explicite donne une indication immédiate sur le type testé. En particulier, on ne doit pas utiliser le test implicite pour les pointeurs. Il faut donc plutôt utiliser :

```
Ligne* ligne;
// …
if (ligne != 0) {
      // …
}
```

plutôt que

```
Ligne* ligne;
// …
if (ligne) {
      // …
}
```

Le cas des variables booléennes est une exception à cette règle. Dans ce cas, il est tout à fait acceptable, et même préférable, de faire un test implicite, puisque le test est lui-même une expression booléenne :

```
bool estActif = false;
// …
if (estActif) {
      // …
}
```

50 Les variables devraient être gardées vivantes le moins longtemps possible.

Il est plus facile de contrôler les effets directs et les effets secondaires d'une variable si on garde les opérations sur cette dernière à l'intérieur d'une petite portée. En pratique, on essaie de limiter la portée d'une variable en la déclarant dans le bloc où elle est utilisée :

```
// RECOMMANDÉ :
// …
for (int i = 0; i < 100; i++) {
      // …
}

// À ÉVITER :
// …
int i;
// …
for (i = 0; i < 100; i++) {
      // …
}

// RECOMMANDÉ :
// …
bool estActif = false;
while (estActif) {
      // …
      int uneVariable = 0;    // Cette variable n'est pas
                              // utilisée en dehors du while.
      // …
}
```

2.5.3. Les boucles

51 Seuls les énoncés de contrôle de boucle doivent être inclus dans la construction du « `for` ».

```
somme = 0; // À ÉVITER : for (i = 0, somme = 0; i < 100; i++)
for (i = 0; i < 100; i++)
      somme += valeur[i];
```

Cela augmente la maintenabilité et la lisibilité. On distingue clairement ce qui contrôle la boucle de ce qui est contenu dans la boucle.

52 Les variables de boucle devraient être initialisées immédiatement avant leur utilisation au sein d'une boucle.

```
bool estFini = false;
while (!estFini) {
      // …
}

// À ÉVITER :
bool estFini = false;
// …
// …
while (!estFini) {
      // …
}
```

53 L'utilisation de boucles « `do-while` » devrait être évitée.

Les boucles « `do-while` » sont moins lisibles que les boucles « `while` » et les boucles « `for` », car la condition est située au bas de la boucle. Le lecteur doit lire la boucle en entier pour comprendre la portée de la boucle.

De plus, les boucles « `do-while` » ne sont pas nécessaires. N'importe quelle boucle « `do-while` » peut facilement être réécrite en une boucle « `while` » ou une boucle « `for` ». Réduire le nombre de constructions différentes améliore la lisibilité.

54 L'utilisation de « break » et « continue » dans les boucles devrait être évitée.

Ces énoncés devraient seulement être utilisés s'ils augmentent la lisibilité par rapport à leurs équivalents structurés. Le cas typique où on serait tenté d'utiliser ces instructions est celui où une situation exceptionnelle se produit à l'intérieur d'une boucle. Supposons par exemple qu'une méthode peut échouer:

```
for (iterateur = unConteneur.begin();
     iterateur != unConteneur.end();
     ++iterateur) {
      // …
      if (unObjet.uneMethode() == false)
           break;
      // …
}
```

Avant d'utiliser un « break » dans une telle situation, il faut d'abord se demander s'il n'est pas plus approprié de lancer une exception. Il y a, par contre, un cas où le « break » s'impose. Il s'agit de la boucle infinie :

```
while (true) {
      // …
      if (estTermine)
           break;
      // …
}
```

55 La forme « `while (true)` » devrait être utilisée pour les boucles infinies.

```
while (true) {
	// …
}

for (;;) { // À ÉVITER!
	// …
}

while (1) { // À ÉVITER!
	// …
}
```

Tester par rapport à « 1 » n'est ni nécessaire ni significatif. La forme « `for (;;)` » n'est pas très lisible et n'indique pas clairement qu'il s'agit d'une boucle infinie.

2.5.4. Les instructions conditionnelles

56 Les expressions conditionnelles complexes doivent être évitées. Introduire plutôt des variables booléennes temporaires, voire de nouvelles méthodes dédiées.

```
bool estFini = (noElement < 0) || (noElement > maxElement);
bool estEntreeRepetee = (noElement == dernierElement);

if (estFini|| estEntreeRepetee) {
      // …
}

// À ÉVITER :
if ((noElement < 0) || (noElement > maxElement)||
     noElement == dernierElement) {}
```

En assignant des variables booléennes aux expressions, le programme s'auto-documente. La condition sera plus facile à lire, à déboguer (possibilité de voir la valeur de chacun des tests individuellement) et à maintenir.

57 Le cas le plus fréquent d'une construction « `if` » devrait être mis dans la partie « `if-then` » et l'exception dans la partie « `else` ».

```
bool estOk = lireFichier(nomFichier);

if (estOk) {
      // …
}

else {
      // …
}
```

Cette norme sert à s'assurer que les cas d'exceptions (c'est-à-dire les moins fréquents) n'entravent pas le cours normal d'exécution. Cela est important pour la lisibilité **ET** pour la performance.

58 Le code conditionnel devrait être mis sur une ligne distincte.

```
if (estFini)
faireMenage();

// À ÉVITER:  if (estFini) faireMenage();
```

Cette règle facilite principalement le débogage. Lorsque le code conditionnel est écrit sur une seule ligne, il n'est pas évident de savoir si le test a échoué ou réussi.

Pour un code final (en production), il est préférable de refactoriser le code précédent en une seule ligne, et ce, pour toutes les instances, **uniquement dans le cas d'une solution clef en main**, non vouée à des changements ultérieurs.

59 Un énoncé de traitement ne doit pas se trouver à l'intérieur d'un conditionnel.

```
File* fileHandle = open(fileName, "w");
if (fileHandle != 0) {
      // …
}

// À ÉVITER :
if ((fileHandle = open(fileName, "w")) != 0) {
      // …
}
```

Cette règle permet d'offrir une meilleure lisibilité.

2.5.4. Divers

60 L'utilisation de nombres « magiques » dans le code doit être évitée. Les nombres autres que « `0` » et « `1` » peuvent être déclarés comme constantes nommées, que l'on déclare au début du fichier ou dans un fichier de configuration exporté (« un fichier de constantes »).

```
const int MAX = 15;
```

Si le nombre n'a pas une signification évidente, la lisibilité est améliorée par l'introduction d'une constante nommée. En regroupant ces constantes, un autre objectif visé est de faciliter la maintenance du code. S'il faut modifier la valeur de la constante définie à un endroit bien précis, on n'a pas à rechercher et à remplacer toutes ses occurrences dans le code. Cela augmente la **portabilité** et la **maintenabilité** d'un projet.

61 Les nombres constants à virgule flottante doivent toujours être écrits avec un point décimal et au moins une décimale.

```
double total = 0.0;           // À ÉVITER : double total = 0;
double vitesse = 3.0e8;       // À ÉVITER : double speed = 3e8;
double sum;                   // À ÉVITER : double sum   = 0.;
// …
sum = (a + b) * 10.0;
```

Cette norme respecte la nature différente des entiers et des nombres à virgule flottante. Mathématiquement, les deux modèles sont complètement différents et sont des concepts incompatibles. De plus, comme on le voit dans le dernier exemple, cela fait ressortir le type de la variable à laquelle on assigne un nombre, à un point dans le code où le type n'est peut-être pas évident.

62 Les nombres constants à virgule flottante doivent toujours être écrits avec un chiffre avant le point décimal.

```
double total = 0.5; // À ÉVITER : double total = .5;
```

Le système de nombres et d'expressions de *C++* est emprunté des mathématiques et on devrait adhérer autant que possible ses conventions. De plus, « `0.5` » est plus lisible que « `.5` »; il est beaucoup plus difficile de confondre « `0.5` » avec l'entier « `5` ».

63 L'instruction « goto » ne doit pas être utilisée.

L'instruction « goto » viole l'idée de la programmation structurée. « goto » ne devrait être considérée que dans des cas très rares (par exemple, pour sortir d'une structure profondément imbriquée) et seulement si l'équivalent structuré est moins lisible. Néanmoins, tout développeur devrait se poser de sérieuses questions de refactorisation de code avant d'employer unetelle instruction « de secours ».

2.6. DISPOSITION ET COMMENTAIRES

2.6.1. Disposition

64 L'indentation de base doit être de 2 ou 4 espaces (d'une demi-tabulation à une tabulation).

```
for (i = 0; i < nElements; i++)
   a[i] = 0;
```

L'indentation est utilisée pour mettre en relief la structure logique du code. Une indentation d'une espace est trop petite pour accomplir cela. Une indentation de plus de quatre espaces rend le code très imbriqué difficile à lire et augmente la probabilité de la nécessité de couper des lignes. Une fois qu'on a choisi le nombre d'espaces d'indentation, il faut bien entendu l'appliquer de manière homogène à tout le code.

65 **La disposition des blocs doit être telle qu'illustrée dans les exemples 1 et 2 plus bas, et ne doit pas être comme l'exemple 3. Les blocs des déclarations de classes, d'interfaces et de méthodes doivent utiliser la disposition de l'exemple 2.**

```
// Exemple 1
while (!estTermine) {
      faireQuelqueChose();
      fini = aEncoreAFaire();
}

// Exemple 2
while (!estTermine)
{
      faireQuelqueChose();
      fini = aEncoreAFaire();
}

// Exemple 3
while (!estTermine)
      {
            faireQuelqueChose();
      fini = aEncoreAFaire();
      }
```

L'exemple **3** ajoute un niveau d'indentation additionnel qui ne fait pas ressortir la structure logique aussi bien que les exemples **1** et **2**. **La disposition recommandée est celle de l'exemple 1.** Quelle que soit la norme que vous adopterez, assurez-vous de l'utiliser de manière homogène dans tout le code.

66 Les déclarations de classes doivent avoir la forme suivante :

```
class MaClasse: public ClasseDeBase
{
public:
	// …
protected:
	// …
private:
	// …
};
```

Cela découle, en partie, de la règle de la disposition des blocs.

67 Les définitions de méthodes doivent avoir l'une des formes des formes suivantes :

```
// Forme 1
void maMethode()
{
	// …
}

// Forme 2
void maMethode() {
	// …
}
```

Cela découle, en partie, de la règle de la disposition des blocs.

68 Les énoncés de type « `if-else` » doivent respecter l'une des deux normes suivantes :

```
      // Style recommandé
 if (condition) {          if (condition)       if (condition) {
            // …           {                        // …
 }                                 // …       }
                           }                  else if (condition)
                           else {             {
                                 // …              // …
                           }                  }
                                              else {
                                                   // …
                                              }
```

```
      // Style alternatif
if (condition)          if (condition)          if (condition)
{                       {                       {
            // …                 // …                 // …
}                       }                       }
                        else                    else if (condition)
                        {                       {
                                 // …                 // …
                        }                       }
                                                else
                                                {
                                                      // …
                                                }
```

Cela découle en partie de la règle sur la disposition des blocs. Par contre, on pourrait discuter de la possibilité de mettre une clause « `else` » sur la même ligne que l'accolade fermante de la clause « `if` » ou « `else` » précédent :

```
     if (condition) {
           // …
     } else {
           // …
     }
```

(...)

(...)

L'approche choisie est considérée meilleure, car chaque partie de l'énoncé « `if-else` » est écrite sur des lignes différentes du fichier. Il est donc plus facile de manipuler l'énoncé, par exemple pour déplacer une clause « `else` ».

69 Une boucle « `for` » doit respecter l'une des deux normes suivantes :

```
// Style recommandé
for (initialisation; condition; mise à jour) {
      // …
}
// Style alternatif
for (initialisation; condition; mise à jour)
{
      // …
}
```

Cela découle de la règle de la disposition des blocs.

70 Un énoncé « `for` » vide devrait avoir la forme suivante :

```
for (initialization; condition; update)
      ;
```

Cela met en évidence le fait que l'énoncé est vide et que cela est intentionnel.

71 L'énoncé « `while` » doit respecter l'une des deux normes suivantes :

```
// Style recommandé
while (condition) {
      // …
}
```

```
// Style alternatif
while (condition)
{
      // …
}
```

Cela découle de la règle de la disposition des blocs.

72 L'énoncé « do-while » doit respecter l'une des deux normes suivantes :

```
// Style recommandé
do {
      // …
} while (condition)
```

```
// Style alternatif
do
{
      // …
} while (condition)
```

Cela découle de la règle de la disposition des blocs.

73 L'énoncé « switch » doit respecter l'une des deux normes suivantes :

```
// Style recommandé
switch (var) {
	case ABC:
		// …
		// Fallthrough
	case DEF:
		// …
		break;
	case XYZ:
		// …
		break;
	default:
		// …
		break;
}
```

```
// Style alternatif
switch (var)
{
	case ABC:
		// …
		// Fallthrough
	case DEF:
		// …
		break;
	case XYZ:
		// …
		break;
	default:
		// …
		break;
}
```

Chaque mot-clef « `case` » est indenté par rapport à l'énoncé « `switch` » lui-même. Cela fait ressortir davantage l'énoncé « `switch` ». Le commentaire « *Fallthrough* » explicite devrait être ajouté chaque fois qu'un énoncé « `case` » n'a pas d'énoncé « `break` ». L'oubli de l'énoncé « `break` » est une erreur commune, alors on doit indiquer clairement qu'il est intentionnel de ne pas en avoir dans certains cas.

74 Un énoncé « `try-catch` » doit respecter une des deux normes suivantes:

```
// Style recommandé
try {
      // …
}
catch (Exception& exception) {
      // …
}
```

```
// Style alternatif
try
{
      // …
}
catch (Exception& exception)
{
      // …
}
```

Cela découle partiellement de la règle de la disposition des blocs. La discussion sur les accolades fermantes pour « `if-else` » s'applique aussi aux énoncés « `try-catch` ».

75 Un énoncé « `if-else` », « `for` » ou « `while` » simple peut être écrit sans accolades.

```
if (condition)
      // …

while (condition)
      // …

for (initialisation; condition; mise à jour)
      // …
```

Les accolades sont généralement un élément du langage qui groupe plusieurs énoncés. Elles sont donc, par définition, superflues autour d'un seul énoncé. Un argument courant contre cette syntaxe est que l'ajout d'un énoncé additionnel va briser le code si on ne rajoute pas également les accolades (à revisiter au sein de vos méthodes dites « **finales** »).

2.6.2. Espaces blancs

77 **Les opérateurs conventionnels doivent être encadrés d'espaces. Les mots réservés en *C++* doivent être suivis d'une espace. Les virgules doivent toujours être suivies d'une espace. Les deux points (« : ») doivent être suivis d'une espace. Les points-virgules (« ; ») des énoncés « `for` » doivent être suivi d'une espace.**

```
a = (b + c) * d;
// À ÉVITER : a=(b+c)*d

while (true) {
// À ÉVITER : while(true){

faireQuelqueChose(a, b, c, d);
// À ÉVITER: faireQuelqueChose(a,b,c,d);

case 100:

for (i = 0; i < 10; ++i) {
// À ÉVITER : for(i=0;i<10;i++){
```

Ceci permet de faire ressortir les composantes individuelles des énoncés et améliore la lisibilité. Il est difficile de donner une liste complète de l'utilisation suggérée des espaces dans le code *C++*. Les exemples ci-dessus devraient donner une idée générale des intentions.

78 Les noms des méthodes peuvent être suivis d'une espace lorsqu'ils sont suivis d'un autre nom.

```
faireQuelqueChose (fichierCourant);
```

Cela met en relief les noms et peut améliorer la lisibilité. Lorsqu'une méthode n'est pas suivie d'un autre nom, l'espace peut être omis (« `faireQuelqueChose()` »), car il n'y a aucun doute sur le nom dans ce cas.

Une alternative à cette approche est d'ajouter un espace après la parenthèse d'ouverture. Ceux qui adhèrent à cette approche laissent en général un espace avant la parenthèse de fermeture : « `faireQuelqueChose( fichierCourant );` ». Ceci fait ressortir les noms individuels, mais l'espace avant la parenthèse de fermeture est artificiel, et sans cet espace, l'énoncé semble plutôt asymétrique (« `faireQuelqueChose( fichierCourant);` »).

Peu importe la convention employée, assurez-vous de vous y tenir pour l'entièreté des fichiers de votre projet.

79 Les unités logiques à l'intérieur d'un bloc devraient être séparées par une ligne vide.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Augmente la lisibilité en ajoutant de l'espace blanc entre les unités logiques. Chaque unité est souvent introduite par un commentaire comme on le voit dans l'exemple ci-dessus.

80 Les méthodes devraient être séparées les unes des autres par deux ou trois lignes vides.

En séparant les méthodes d'un espace plus grand que celui utilisé à l'intérieur de celle-ci, les méthodes sont faciles à distinguer à l'intérieur de la classe. N'oublier pas d'uniformiser pour l'entièreté de vos fichiers d'un même projet.

81 Les variables peuvent être alignées sur la gauche dans les déclarations.

```
AsciiFile* fichier;
int        nPoints;
double     x, y;
```

Cela permet d'améliorer la lisibilité. Les variables sont plus faciles à localiser à partir des types en raison de l'alignement.

82 Les énoncés devraient être alignés partout où cela améliore la lisibilité.

```
if      (a == lowValue)    computeSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential    * oilDensity     ) / constant1 +
        (depth              * waterDensity ) / constant2 +
        (zCoordinateValue  * gasDensity   ) / constant3;

minPosition     = computeDistance(min, x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (phase) {
    case PHASE_OIL   : text = "Oil";   break;
    case PHASE_WATER : text = "Water"; break;
    case PHASE_GAS   : text = "Gas";   break;
}
```

Il existe de nombreux cas pour lesquels de l'espace blanc peut être ajouté pour augmenter la lisibilité, même si cela va à l'encontre des recommandations. Il est difficile de donner des règles générales pour l'alignement de code, mais les exemples ci-haut devraient fournir l'idée générale. En bref, n'importe quelle construction qui augmente la lisibilité devrait être permise.

2.6.3. Commentaires

83 Le code difficile à comprendre ne devrait pas être commenté, mais bien réécrit.

En général, l'utilisation des commentaires devrait être minimisée en rendant le code auto-documenté, grâce à des choix de noms judicieux et une structure logique explicite.

84 Tous les commentaires devraient être écrits uniquement en anglais ou uniquement en français.

Dans un environnement international, l'anglais est la langue de choix.

85 Utiliser « // » pour tous les commentaires, même pour les commentaires de plus d'une ligne.

```
// Commentaire sur plus
// d'une seule ligne.
```

Cela nous assure de toujours pouvoir commenter une section d'un fichier en utilisant « /* */ », par exemple pour déboguer un problème, etc. Il devrait toujours y avoir un espace entre le « // » et ledit commentaire. De plus, les commentaires devraient toujours commencer par une lettre majuscule et se terminer par un point.

86 Les commentaires devraient être indentés par rapport à leur position dans le code.

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

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

On veut éviter que les commentaires ne brisent la structure logique du code.

3. POUR ALLER PLUS LOIN…

À la lumière des nombreux concepts et techniques abordés tout au long de ce guide de codage *C++*, il est impératif de rappeler l'importance cruciale de la qualité logicielle.

Le ***C****onsortium for* ***I****T* ***S****oftware* ***Q****uality* (***CISQ***) a élaboré des normes de mesure automatisée du code source pour la fiabilité, la sécurité, la performance et la maintenabilité. Ces normes, après avoir été reconnues par l'***O****bject* ***M****anagement* ***G****roup* (***OMG***), ont été élargies pour englober les faiblesses des logiciels informatiques et embarqués, aboutissant à la norme sur les mesures de qualité automatisées du code source.

Cette dernière a ensuite été soumise et adoptée par l'Organisation internationale de normalisation, devenant ainsi la norme **ISO/IEC 5055:2021**. Il est donc essentiel pour tout professionnel du codage *C++* de ne pas uniquement maîtriser la syntaxe et les structures du langage, mais aussi de veiller à ce que son travail soit en parfaite adéquation avec les standards internationaux de qualité, tels que définis par l'*ISO 5055*.

4. RÉFÉRENCES

[1] GEOSOFT. « *Geosoft C++ Programming Style Guidelines* ». [En ligne]. Disponible : http://geosoft.no/development/cppstyle.html

[2] S. McConnell. «*Code Complete*», *Microsoft Press*. [En ligne]. Disponible : https://www.microsoftpressstore.com/store/code-complete-9780735619678

[3] M. Henricson & E. Nyquist. « *Programming in C++*», *Rules and Recommendations*, *Ellemtel*. [En ligne]. Disponible : https://www.ktverkko.fi/~msmakela/software/Ellemtel-rules-mm.html

[4] NASA. « *C / C++ / Java Coding Standards* ». [En ligne]. Disponible : https://ntrs.nasa.gov/citations/20080039927

[5] « *Doxygen Documentation System*». [En ligne]. Disponible : https://www.doxygen.nl/index.html

[6] K. Gabryelski. « *Wildfire C++ Programming Style* », *Wildfire Communications Inc.* [En ligne]. Disponible : http://www.literateprogramming.com/wildfire.pdf

[7] T. Hoff. «*C++ Coding Standard* ». [En ligne]. Disponible : http://www.literateprogramming.com/toddhoff.pdf

[8] GOOGLE. «*Google C++ Style Guide* ». [En ligne]. Disponible : https://google.github.io/styleguide/cppguide.html

[9] « C++ Core Guidelines ». [En ligne]. Disponible : https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines