Normes de programmation
IFT159

Alex Boulanger
Felix-Antoine Ouellet

Collaborateur:
Gabriel Girard

Version 2.1

1 Introduction

Ce document présente les normes de programmation que les étudiantes et étudiants doivent respecter lors de la remise de travaux pratiques dans le cadre du cours Analyse et programmation (IFT159) de l'Université de Sherbrooke.

2 Documentation

Cette section décrit les exigences de documentation des fichiers sources.

2.1 Doxygen

Le format des commentaires doit respecter le format Doxygen (www.doxygen.org). Il doit être possible de générer automatiquement des fichiers de documentation à l'aide d'un outil acceptant le format Doxygen.

2.2 Caractères d'accentuation

Les commentaires ne doivent pas comporter de caractères accentués(à, é, è, ô, etc.) Il faut remplacer les accents par la lettre équivalente (à devient a, é devient e, è devient e, etc.).

2.3 En-tête de fichiers sources

On doit pouvoir retrouver les éléments suivants tout en haut d'un fichier source:

1. Nom du fichier;
2. Noms et matricules des auteurs du fichier;

3. Courte description du contenu du fichier.

Concrètement, il est attendu de voir une en-tête de la forme suivante:

```
/**
* \file mario.cpp
* \author Felix-Antoine Ouellet 55 555 555
* \author Alex Boulanger 66 666 666
* \brief Ce fichier contient l'implementation d'un
*        jeu de Mario Bros. On y retrouve toutes les
*        fonctions necessaires pour traiter les
*        commandes entrees au clavier et mettre a jour
*        l'ecran de jeu en fonction des commandes
*        entrees.
*/
```

2.4 En-tête de fonctions

Une en-tête de fonction doit comporter les éléments suivants dans l'ordre dans lequel ils sont énoncés.

1. Description brève de la fonction;
2. Description des paramètres dans l'ordre qu'ils sont déclarés;
3. Description du retour de la fonction si son type de retour n'est pas `void`.

L'en-tête ci-dessous est un exemple de la manière dont il faut procéder.

```
/**
* \brief Tente d'extraire un nombre en virgule
*        flottante d'une chaine de carateres
* \param[in]  Chaine de caractere representant un
*             nombre en virgule flottante
* \param[out] Reference ou mettre le nombre qu'on tente
*             d'extraire de la chaine de caracteres
* \return L'extraction a reussie?
*/
bool essaieExtraire(string laString, float & leNombre)
```

2.5 En-tête de types abstraits

Une en-tête de structure ou de classe doit comporter une brève description du type déclaré. L'en-tête ci-dessous est un exemple de la manière dont il faut procéder.

```
/**
* \brief Arbre binaire de recherche automatiquement
```

```
*         equilibre. Dans cet arbre, les hauteurs des
*         deux sous-arbres d'un meme noeud different
*         d'au plus un.
*/
class ArbreAVL { };
```

2.6 Type de commentaires à même le code

Si certaines manoeuvres dans le code pourraient paraître obscures à un tiers parti (correcteur ou autre programmeur), l'auteur du code a la responsabilité de les documenter. À cette fin, il peut utiliser un des deux types de commentaires suivants. Si le commentaire peut tenir sur une ligne, il s'écrira de la manière suivante:

```
// Verifie si x est une puissance de 2
bool puissanceDeDeux = ((x != 0) && !(x & (x - 1)));
```

Dans le cas contraire, le commentaire devra être écrit d'une telle façon:

```
/* Tant que l'instruction ne correspond pas a
   une instruction de branchement */
while ((instruction & 0xF0) != 0x10)
{
    // ...
}
```

Finalement, un commentaire peut être insérer après une déclaration de variable pour préciser sa nature.

```
int angleRotation; // Angle de rotation en radians
```

3 Nomenclature

Cette section décrit les bonnes pratiques de nomenclature à utiliser dans les fichiers de code. Ces pratiques existent pour aider les étudiants ainsi que les assistants à l'enseignement à pouvoir identifier rapidement des problèmes simples et, parfois, subtils.

3.1 Variables

Les noms des variables commencent toujours par une minuscule et doivent être des noms communs qui reflètent leurs utilisations. Par exemple, pour représenter une variable contenant un angle, le nom devrait être

```
    int angle;
```

Par contre, souvent un seul mot ne peut représenter précisément la variable. Dans ce cas, les mots sont séparés par des majuscules. Par exemple, pour

représenter un angle de rotation (qui est beaucoup plus précis que la première variable nommée `angle`), le terme à utiliser serait

```
int angleRotation;
```

Il y a beaucoup d'abréviation utile à connaitre. Par exemple, le terme *compteur* est fortement utilisé dans les structures répétitives et est généralement dénoté par `cpt`.

Un terme également utilisé souvent est *nombre* qui est souvent écrit `nb`.

```
int cptColonnes;  // un indicateur sur une colonne
int nbEtudiants; // le nombre d'etudiants
```

Il faut éviter le plus possible les noms de variables simples comme, par exemple, `i`, `j`, `k`. Dans une majorité de cas, ces noms peuvent être remplacés par des noms significatifs.

Ce style de séparation des mots par des majuscule est nommé *Camel Case*.

3.2 Constantes

A la différence d'une variable, une constante ne change jamais de valeur. Pour indiquer cette différence, les constantes sont toujours identifiées avec des noms en majuscules séparés par des soulignements de lignes.

```
const int REPONSE_UNIVERSELLE = 42;
```

Les constantes reflètent des valeurs simples et leurs noms doivent être simples. Par exemple :

```
const float PI = 3.14f;
const double EULER = 2.71828;
const int LONGUEUR_ECRAN = 1024;
const int HAUTEUR_ECRAN = 768;
```

Ce style de séparation des mots par des soulignements est nommé *Snake case*.

3.3 Fonctions

Une fonction représente une action. Le nom doit clairement indiquer la tâche. Contrairement à une variable, le nom d'une fonction commence par un verbe. Par exemple, dans le cas où on désire une fonction qui vérifie si un conteneur est vide :

```
bool estVide(); // Precis et clair
```

Simplement en lisant le nom de la fonction, on sait qu'elle retourne vrai si le conteneur «est vide».

On pourrait également avoir une autre fonction qui fait le même travail

```
bool vide();
```

Contrairement au premier exemple, la fonction `bool vide();` manque de précision vis-à-vis l'action qu'elle effectue: Vérifie-t-elle si l'état est vide? Où bien vide-t-elle le conteneur et retourne un code de succès? Pour éviter ces questionnements, utilisez un nom précis.

Comme dans le cas des variables, on sépare chaque mot du nom par des majuscules. De plus, le nom d'une fonction commence toujours avec une minuscule.

3.4 Classes

Les classes sont des types complexes créées par les programmeurs. Le nom d'une classe doit correspondre à l'entité qu'elle représente. Par exemple, une classe `Point` représente des points.

Le nom d'une classe commence toujours par une majuscule et les mots suivants respectent la notation *Camel Case*. Par exemple:

```
    class Point;
    class LignePolygone; // une ligne d'un polygone
```

4 Formatage

4.1 Organisation des inclusions de fichiers

Lorsqu'il faut inclure des fichiers dans le fichier courant, il faut d'abord inclure les fichiers propres au projet puis les fichiers de bibliothèques externes au projet. De plus, dans chacun de ces regroupements, l'ordre d'inclusion doit être l'ordre alphabétique sauf lorsque cela empêche la compilation pour cause de référence circulaire. Par exemple:

```
#include "MaClasse.h"
#include "MaStruct.h"

#include <algorithm>
#include <iostream>
#include <vector>
```

4.2 Organisation des fonctions

La première fonction définit dans le fichier principal est le `int main()`. Par la suite, la définition des fonctions déclarées pour la première fois dans le `main` doivent suivre.

```
int main()
{
    // declaration des fonctions a utilisees
    void toto();
```

```
    // traitement
    toto();
}

// commentaires de la fonction
// ...
void toto()
{
    // declaration des fonctions utilisees par toto
    ...

    // traitement
    ...
}
```

Ensuite, viennent les fonctions déclarées pour la première fois dans `toto`. Et ainsi de suite.

4.3 Organisation à l'intérieur d'une fonction

L'organisation à l'intérieur d'une fonction doit respecter un ordre précis:

1. Déclaration des fonctions utilisées;
2. Déclaration des constantes utilisées;
3. Déclaration des variables utilisées;
4. Traitement à effectuer;
5. Retour de la fonction.

L'exemple ci-dessous illustre ce qu'il est attendu de l'étudiant dans un contexte minimaliste.

```
int main()
{
    // declaration de la fonction a utiliser
    float operation(int[], int, int);

    // declaration de la constante
    const int TAILLE = 10;

    // declaration des variables
    int nbImporant;
    int tableau[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
```

```
    // traitement
    cout << "Entrez un nombre: ";
    cin >> nbImportant
    cout << endl << "Le resultat est: "
         << operation(tableau, nbImportant, TAILLE)
         << endl;

    // Retour sans probleme
    return 0;
}
```

4.4 Tabulation et indentation

Il est important que le programmeur configure son environnement de travail de telle sorte qu'une tabulation soit équivalente à 4 espaces.

4.5 Mise en forme

La mise en forme d'un programme inclut le placement des espaces, des retours à la ligne, des parenthèses et des accolades.

Par soucis de respect de la mise en forme et de lisibilité du code, les étudiants sont appelés à conformer leur code aux exemples ci-dessous.

Le point le plus important à remarquer dans les exemples suivants est l'utilisation de l'indentation. Ainsi, on constatera que l'indentation du code n'est augmentée que lorsque l'on entre dans une nouvelle portée, c'est-à-dire entre une paire d'accolades. Le fait d'augmenter l'indentation d'une partie du code pour démarquer une étape de calcul, comme dans l'exemple ci-dessous, est donc considéré comme une mauvaise pratique.

```
// Etape 1
action1();

    // Etape 2
    action2();

        // Etape 3
        action3();
```

4.5.1 Variables

```
int monEntier1;
int monEntier2 = 42;
```

Pas soucis de lisibilité, les étudiants ne doivent pas déclarer plusieurs variables sur la même ligne.

```
// NE PAS FAIRE
int monEntier1, monEntier2 = 42;
```

De plus, encore par soucis de lisibilité, il convient de déclarer les variables comme dans le premier exemple et non comme dans l'exemple suivant.

```
// NE PAS FAIRE
int monEntier1,
    monEntier2 = 42;
```

4.5.2 Constantes

```
const int LONGUEUR_ECRAN = 1024;
```

Les constantes doivent toujours être déclarées de manière locale sauf lorsqu'elle sont utilisées dans plusieurs fonctions. Dans ce cas, elles doivent être déclarées de manière globale.

4.5.3 Fonctions

```
void afficherNombre(int nb)
{
    cout << "Le nombre est: " << nb << endl;
    return;
}
```

Pour la déclaration des fonctions, les règles à appliquer sont les mêmes que celles que régissent la déclaration de constantes.

4.5.4 Classes et structures

```
class MaClasse
{
private:
    // section des attributs
    int monEntier;
    string maString;

public:
    // section des constructeurs - destructeur
    MaClasse();
    ~MaClasse();
};
```

Il est important de noter que le mot-clé `private` est spécifié même si par défaut les données et fonctions membres d'une classe sont privées par défaut. Il en serait de même pour une structure et le mot-clé `public`.

4.5.5 Conditionnels - If

```
if ((coordX < 5 && coordX > -5) || coordY < 5
    || coordZ > 10)
{
    action1();
}
else
{
    action2();
}
```

Notez que si l'expression conditionnelle est trop longue pour être affichée sur une seule ligne, un retour à la ligne doit être insérer de telle sorte que la ligne suivante commence par un opérateur logique.

S'il y plus de deux branches possibles, le programmeur est appelé à utiliser la structure `if()...else if()...else` comme suit:

```
if (coordX < 5)
{
    action1();
}
else if (coordX > 10)
{
    action2();
}
else
{
    action3();
}
```

On rappelle qu'il est la responsabilité du programmeur de réfléchir si une série de `if()...else if()...else` pourrait être remplacé par un `switch`.

Finalement, une imbrication de `if` ressemble à ceci:

```
if (coordX < 5)
{
    if (coordY < 5)
    {
        action1();
    }
    else
    {
        action2();
    }

}
```

4.5.6 Conditionnels - Switch

```
switch (varInt)
{
    case 0:
    {
        // ...
        break;
    }
    case 42:
    {
        // ...
        break;
    }
    default:
    {
        // ...
        break;
    }
}
```

4.5.7 Répétitions - While

```
while (i < 10)
{
    action(i);
}
```

4.5.8 Répétitions - Do While

```
int i = 0;
do
{
    action(i);
} while (i < 10)
```

4.5.9 Répétitions - For

```
for (int indice = 0; indice < taille; indice += 1)
{
    tableau[indice] += 10;
}
```

4.5.10 Répétitions - Range-Based For

```
for (int valeur : tableau)
{
    cout << "Valeur: " << valeur << " ";
}
```

5 Erreurs communes et mauvaises pratiques

5.1 Switch sur une variable booléenne

Compte tenu qu'une variable booléenne ne peut prendre que deux valeurs (soit `true` soit `false`), il n'est en aucun cas pertinent de l'utiliser dans un `switch`. L'exemple ci-dessous ne doit donc jamais être retrouvé dans le code d'un étudiant

```
switch (variableBool)
{
    case true:
    {
        // ...
        break;
    }
    case false:
    {
        // ...
        break;
    }
    default:
    {
        // I forgot to brain
        break;
    }
}
```

5.2 Utilisation des variables booléennes

Il est attendu des étudiants qu'ils évitent d'écrire du code semblable

```
if (variableBool == true)
{
    return true;
}
else
{
    return false;
}
```

Dans des cas semblables, ils devraient travailler directement avec la variable booléenne comme ceci:

```
return variableBool;
```

Une mauvaise pratique moins grave, mais tout de même à éviter, est de faire une comparaison directe entre une variable booléenne et les valeurs `true` ou `false`.

```
if (variableBool == true)
{
    // Actions quelconques
}
```

La bonne pratique à employer dans ce cas est d'utiliser directement la variable booléenne.

```
if (variableBool)
{
    // Actions quelconques
}
```

5.3 Affectation dans une structure conditionnelle

Il est important de se rappeler que l'opérateur de comparaison en C++ est `==`. L'oubli de ce détail risque fort de produire des comportements inattendus.

```
int varInt = 0;
while (varInt < taille)
{
    if (varInt = 0)
    {
        calculImportant();
    }
    else
    {
        calculQuelconque();
    }
    varInt++;
}
```

Dans l'exemple ci-dessus, le fait d'affecter 0 à `varInt` au lieu d'effectuer une comparaison va faire en sorte que la boucle sera infinie.