

Programmation

- Standard de codage
- Documenter son code
- Tests unitaires démo pratique

Standard de codage

Définition (Wikipédia)

Les règles de codage sont un ensemble de règles à suivre pour uniformiser les pratiques de développement logiciel, diffuser les bonnes pratiques de développement et éviter les erreurs de développement “classiques” au sein d’un groupe de développeurs. Les règles de codage s’articulent autour de plusieurs thèmes, les plus courants étant :

1. Le nommage et l’organisation des fichiers du code source
2. Le style d’indentation
3. Les conventions de nommage, ou règles de nommage
4. Les commentaires et documentation du code source
5. Recommandations sur la déclaration des variables
6. Recommandations sur l’écriture des instructions, des structures de contrôle et l’usage des parenthèses dans les expressions.

Ressources documentaires

[Wikipedia](#)

[GNU Coding standard](#)

[clang_format](#)

[clang](#)

[LLVM](#)

Indentation du code

En informatique : l’indentation consiste en l’ajout de tabulations ou d’espaces dans un fichier, pour une meilleure lecture et compréhension du code

Indenter automatiquement votre code

- Vous pouvez indenter automatiquement sous Qt le code source par les raccourcis suivants

CTRL+A sélectionne le texte

CTRL+I formate automatiquement la sélection dans QT

Lignes orphelines

Les lignes sans codes, ou lignes orphelines doivent être supprimées, sauf si elles servent à délimiter les blocs d'instructions (comme par exemple entre la déclaration des variables et le début des instructions)

Conventions de nommage à respecter

Déclaration des variables

- Les variables doivent avoir des noms représentatifs de leur contenu

****Exemple (je veux déclarer un entier contenant des notes) ****

```
int notes;
```

■

- Pour les tableaux, je fais précéder le nom du tableau de `tab`

****Exemple (Je veux déclarer un tableau contenant des notes (comme 14,34 ou 5,75) ****

```
float tabNote[10];
```

■

- Pour les pointeurs, je fais précéder le nom du pointeur de `ptr`

```
int *ptrNote = nullptr;
```



Convention de nommage

Je respecte la notation lowerCamelCase pour mes variables et fonctions

- Premier mot en minuscule
- Les mots suivants avec une majuscule

Variables et fonctions

****Exemple (Je veux déclarer un entier contenant un nombre de livres) ****

```
int nombreLivre;
```



****Exemple (Je veux déclarer une fonction pour sauver une image en niveau de gris) ****

```
void sauverImageNiveauGris();
```



Les macro

- Les noms des macro doivent être en majuscules

****Exemple, une macro qui calcule le max de deux nombres ****

```
#define MAX(x,y) ((x)>(y)?(x):(y))
```



Les structures

- Le type de la structure doit commencer par une majuscule

****Exemple, j'ai une structure `Pixel` ****

```
typedef struct {  
    unsigned char red;  
    unsigned char blue;  
    unsigned char green;  
} Pixel;
```

- - Je déclarerai une variable du type Pixel comme indiqué ci-dessous

```
Pixel monPixel;
```

■

Les classes

NOM DES CLASSES

- Chaque classe commencera par 'C' suivi du nom de la classe avec la première lettre en majuscule

****Exemple, une classe contenant des informations sur des avions sera déclarée ainsi ****

```
class CAvion {  
    public :  
        //Méthodes publiques  
    private :  
        //Méthodes et attributs privés  
};
```

■

MÉTHODES DE LA CLASSE

- Les méthodes de la classe respectent la convention lowerCamelCase

ATTRIBUTS DE LA CLASSE

- Pour les attributs, je fais précéder le nom de l'attribut de "_"

****Exemple (un attribut contenant un prénom) ****

```
string _prenom;
```



CLASSE CAVION AVEC MÉTHODES ET ATTRIBUTS

```
class CAvion {  
    public :  
        int controlerAssiette();  
    private:  
        float _altitude;  
};
```

Documenter son code

Documentation des fichiers du projet

Chaque fichier **doit** avoir sa documentation au début de celui-ci

Exemple

```
/**
 * @file Nom du fichier
 * @brief Résumé du rôle du module/classe
 * @brief On peut continuer le résumé sur plusieurs lignes
 * @author Auteur du fichier
 * @version v1.0
 * @class Nom de la classe (si POO)
 * @date 01/01/1900
 */
```



Documentation des fonctions ou méthodes

- La documentation doit être écrite dans les fichiers `.h`
- Chaque fonction/méthode **doit** avoir sa documentation avant son prototype

Cas général

```
/**
 * @fn type nomFonction(type parametre1, type parametre2);
 * @brief Résumé de la fonction
```

```
* @param type parametre1 : Le rôle du paramètre 1
* @param type parametre2 : Le rôle du paramètre 2
* @return type : le rôle de la valeur de retour
*/

type nomFonction(type parametre1, type parametre2);
```

■

Exemple : méthode `string findClientName(int idClient);`

```
/**
 * @fn string findClientName(int idClient);
 * @brief trouve le nom du client en fonction de son id
 * @param int idClient : l'id du client à rechercher
 * @return string : nom du client
 */

string findClientName(int idClient);
```

■

Attribut des classes

- Chaque attribut doit être documenté ainsi

Cas général

```
type nom; //!< rôle de l'attribut
```

■

Exemple

```
QSqlDatabase _dbParking; //!< base de données
```

■

Balises supplémentaires

@code

Cette balise permet d'insérer un exemple de code dans la documentation

```
/**
 * @code
 * #include <iostream>
 *
 * int main()
 *{
 * //Your code
 *
 * return 0;
 *}
 * @endcode
 **/
```

■

Doxyfile

On peut ensuite générer la documentation grâce à [doxygen](#)

Si doxygen n'est pas installé sur le PC, exécuter la commande suivante

```
apt install doxygen graphviz doxygen-gui
```

■

Ensuite, on peut générer la documentation

- Soit, en suivant le doxywizard dans le dossier du projet

```
doxywizard
```

-
- Soit en lançant directement la commande (il faut alors avoir un doxyfile cf-ci-après)

```
doxygen Doxyfile
```

■

[Exemple de doxyfile C](#)

[Exemple de doxyfile cpp](#)

Générer automatiquement la documentation avec le pipeline Gitlab CI/CD

- Ajouter le stage ci-dessous à votre fichier .gitlab-ci.yml

```
stages:
  - deploy

pages:
  stage: deploy
  script:
    - doxygen Doxyfile
    - mv html/ public/
  artifacts:
    paths:
      - public
```

■

ToDo

- La documentation générée n'est accessible que depuis notre réseau local

Doc gitlab et doxygen

Tests unitaires démo pratique

1 - Un exemple avec des tests

Considérez l'exemple d'une structure de données "file" (ou *queue* en anglais), tel que les éléments les premiers entrés sont aussi les premiers sortis (First In, First Out) :

<https://git.vainsta.fr/share/queue>.

Ce petit projet se compose de plusieurs fichiers, dont voici une brève description :

- le module *queue* (`queue.c` + `queue.h`)
- un exemple d'utilisation de la queue (`sample.c`)
- un fichier de tests du module *queue* (`test_queue.c`)
- le fichier `CMakeLists.txt` pour compiler ce projet

La compilation du projet et l'exécution des tests se fait de la manière suivante :

```
$ mkdir build ; cd build
$ cmake .. ; make      # compilation
$ make test           # lancement des tests
```

La commande `make test` va lancer l'exécution de tous les tests définis dans `CMakeLists.txt` et qui sont implémentés dans le fichier `test_queue.c`. La fonction `main()` prend en paramètre le nom du test à exécuter (*init_free*, *push*, *pop*, *length*, *empty*) et appelle la fonction de test associée.

Ainsi, l'exécution de la commande ci-dessous exécute un test, qui a pour objectif de vérifier le code de la fonction `queue_length()` de notre module *queue* :

```
$ ./test_queue length      # execution du test "length"
$ echo $?                  # afficher le code de retour de la commande précédente
0                          # 0 => success!
```

On considère qu'un test est réussi (*success*) si et seulement si le code de retour du programme de test est égal à 0, c'est-à-dire que la fonction `main()` renvoie la valeur `EXIT_SUCCESS` (ou 0). Toutes les autres valeurs de retour correspondent à des cas d'erreur !

En résumé :

- 0 ou `EXIT_SUCCESS`, l'exécution du test s'est terminée normalement en ne détectant aucun problème.
- 1 ou `EXIT_FAILURE`, l'exécution du test s'est terminée normalement en détectant un problème.
- Des valeurs supérieures à 128 correspondent à des terminaisons anormales (programme de test tué par un signal), liés à une erreur grave comme un accès illégal à la mémoire (*Segmentation Fault*, 139), une division par zéro (*Floating Point Exception*, 136), ...

Plus l'écriture d'un test est "précise", plus il doit être facile de conclure qu'il y a un bug dans la fonction testée (et non pas dans une autre fonction). L'ensemble des tests doit *couvrir* toutes les fonctions de notre module et être le plus exhaustif possible !

Notez que la commande `make test` va appeler le framework *CTest* (intégré à *CMake*) qui affiche un petit rapport d'exécution de tous les tests...

```
$ make test
Running tests...
Test project /home/orel/Documents/pt2/misc/queue/build
  Start 1: test_queue_new_free
1/8 Test #1: test_queue_new_free ..... Passed    0.00 sec
  Start 2: test_queue_push_head
2/8 Test #2: test_queue_push_head ..... Passed    0.00 sec
  Start 3: test_queue_pop_head
3/8 Test #3: test_queue_pop_head ..... Passed    0.00 sec
  Start 4: test_queue_push_tail
4/8 Test #4: test_queue_push_tail ..... Passed    0.00 sec
  Start 5: test_queue_pop_tail
5/8 Test #5: test_queue_pop_tail ..... Passed    0.00 sec
  Start 6: test_queue_length
6/8 Test #6: test_queue_length ..... Passed    0.00 sec
  Start 7: test_queue_empty
7/8 Test #7: test_queue_empty ..... Passed    0.00 sec
  Start 8: test_queue_clear
8/8 Test #8: test_queue_clear ..... Passed    0.00 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =  0.03 sec
```

2 - Annexes

Un exemple de fonction test

Voici un exemple de test (à compléter) pour la fonction `game_is_empty()` du jeu Takuzu :

```
bool test_is_empty(void)
{
    game g = game_default();
    bool test1 = game_is_empty(g, 0, 0);
    bool test2 = !game_is_empty(g, 5, 5);
    game_delete(g);
    return test1 && test2;
}
```