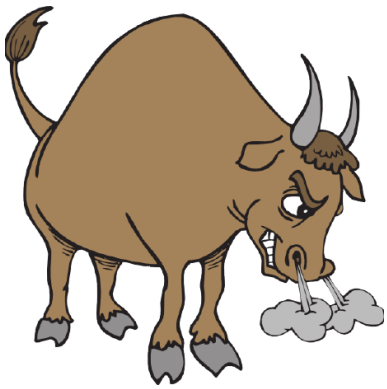

ox223252

Coding Rules

[Fr]

Table des matières

Table des figures

Préface

Avant de rentrer dans le technique il y a quelques petites règles que tout programmeur devrait avoir en tête continuellement :

Tout ce qui est susceptible de mal tourner tournera mal.

Murphy

S'il y a plus d'une façon de faire quelque chose,
et que l'une d'elles conduit à un désastre, alors
il y aura quelqu'un pour le faire de cette façon.

Murphy

Quand quelque chose tourne mal, quelque chose de pire
arrive toujours à ce moment là.

LEM (Loi de l'Emmerdement Maximum)

La catastrophe qui finit par arriver n'est jamais celle
à laquelle on s'est préparé.

Twain

S'il y a la moindre possibilité que ça rate, ça ratera ;
s'il n'y en a aucune, ça ratera quand même.

Finagle

Général

Pour tous les langages il est certaines règles qui devraient être appliquées autant que possible.

Un analyseur de code devrait être utilisé pendant le développement d'un logiciel ou d'une application quelque soit le langage. L'utilisation d'un outil d'analyse syntaxique de code (linter) du type `JsLint` (javascript) permet de vérifier le code afin d'éviter des erreurs ou omissions pouvant provoquer des dysfonctionnements et des difficultés à déboguer.

Une fois le code créé, un logiciel d'analyse dynamique type `gdb` ou `valgrind` (C/C++) devrait être utilisé pour vérifier la stabilité, les fuites mémoires et autres, vous évitant ainsi des omissions potentiellement coûteuses.

Lors de l'écriture d'un programme, vous devez aussi penser à l'écriture de fonctions de tests unitaires, soit manuellement soit avec un outil d'automatisation, ainsi qu'à' l'utilisation d'un logiciel de couverture de code (code coverage), vous permettant de vérifier que vous avez testé toutes les parties importantes de votre code.

Vous augmenterez ainsi la qualité du code et que sa stabilité.

Attention, les règles de codage s'appliquent aux fichiers que vous écrivez pas à ceux des différentes librairies ou sources externes.

Vous pouvez trouver la dernière version de ce document sur Github.

C

3.1 Projet

Nous conseillons (pour ne pas dire nous imposons), de ne pas utiliser d'IDE [1], pour la compilation d'un programme. Un makefile (ou autre outils libre/OpenSource tel Cmake) est conseillé, couplé avec un éditeur de texte.

Nous souhaitons qu'un simple `make` à la base de l'arborescence permette de compiler le code.

3.1.1 Avec un IDE

Bien que nous demandons un travail dans IDE certain préfèrent quand même en utiliser un. L'utilisation d'un IDE permet de prendre des libertés sur le propreté et la gestion du projet. Vous ne devez en aucun cas vous autoriser à du laxisme dans la gestion de vos travaux, dans le cas contraire ils deviendrons rapidement d'une complexité affolante et impossible à suivre ou reprendre pour un nouveau développeur.

3.1.2 Pas d'IDE

Makefile

Le makefile à la racine de votre projet doit comporter comme première directive `all:` qui permet de compiler tout votre projet, ainsi qu'une directive `mrproper` qui doit permettre de supprimer tous les fichiers générés automatiquement (`*.o`, `exec`).

La compilation doit se faire en `-Wall` sans que cela n'affiche le moindre warning car un warning est un bug potentiel.

Un exemple de `makefile` de base :

1. logiciel qui cache des éléments de compilation et autres

```
SOURCE = src
ROOT_DIR = bin
EXEC = execName
CCFLAGS = -Wall
CXXFLAGS = -Wall
CPP_SRC=$(shell find $(SOURCE) -name *.cpp)
CPP_OBJ=$(CPP_SRC:.cpp=.o)
C_SRC=$(shell find $(SOURCE) -name *.c)
C_OBJ=$(C_SRC:.c=.o)
OBJ=$(C_OBJ) $(CPP_OBJ)

all: makedeps $(ROOT_DIR)/$(EXEC)
$(ROOT_DIR)/$(EXEC): $(OBJ)
    $(CXX) -o $@ $^ $(LDFLAGS)
%.o: %.c
    $(CC) -c -o $@ $< $(CCFLAGS)
%.o: %.cpp
    $(CXX) -c -o $@ $< $(CXXFLAGS)
clean:
    rm -f $(shell find $(SOURCE) -name *.o)
    rm -f $(shell find $(SOURCE) -name *.so)
mrproper: clean cleanDump
    rm -f $(ROOT_DIR)/$(EXEC)
makedeps:
    @makedepend -Y $(C_SRC) $(CPP_SRC) 2> /dev/null
    #
```

FIGURE 3.1 – Makefile

Cmake

À faire

3.1.3 Arborescence

Un projet doit contenir un dossier `src` et un `makefile` (ou autre outils équivalent), l'exécutable doit être généré dans un second dossier `bin`. De plus, il peut être adjoint un dossier `doc` qui contiendra la documentation, un dossier `res` qui contiendra les fichiers utile au fonctionnement de l'exécutable (fichiers de config/données/...), `lib` pour les fichiers nécessaires à la compilation, autre que les codes sources.

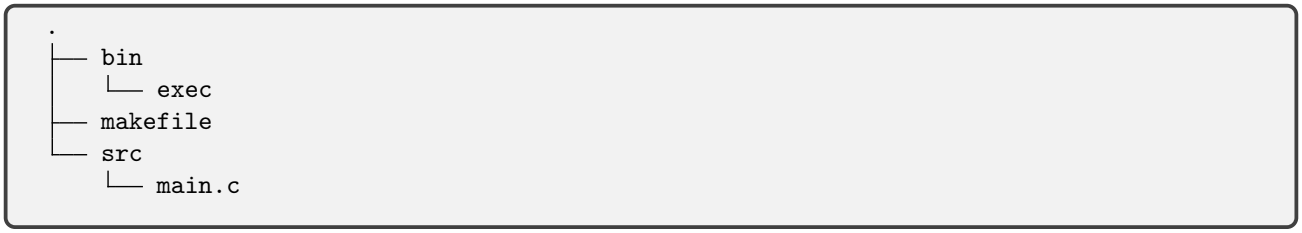

FIGURE 3.2 – Arborescence de configuration du projet

Nous vous conseillons aussi d'avoir un dossier de compilation qui pourra contenir les différentes architectures cible de votre projet. Cela permet de séparer les fichier des compilations et les fichiers sources, de plus cela permet de compiler en parallèle plusieurs architectures (si besoins est).

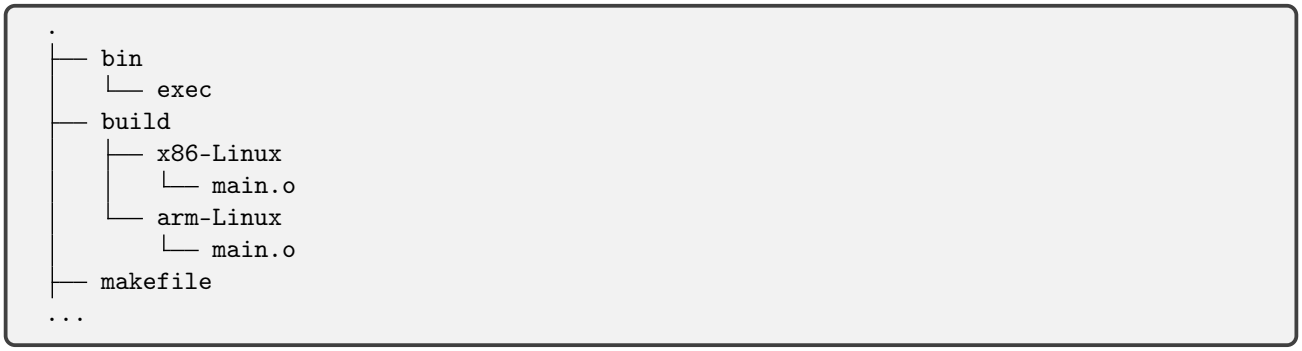

FIGURE 3.3 – Arborescence de compilation du projet

Sous arborescence

Les fichiers sources doivent être dispatchés dans des sous dossiers de façon à séparer les éléments du code en les regroupant par catégorie.

Le nombre de sous répertoires ainsi que leurs profondeur est sans limite. Cependant la compréhension du code doit rester claire.

```
.
└── src
    ├── libs
    │   └── json
    │       ├── json.c
    │       └── json.h
    └── main.c
```

FIGURE 3.4 – Sous arborescence de projet

inclusion / et lib

Beaucoup de gens on la mauvaise manie d'utiliser le flag `-Idir` de gcc qui permet d'inclure des dossiers dans les path utilisé pour trouver les headers. Il est vrai que techniquement parlant c'est fonctionnel, mais cela nuit de façon non négligeable à la lisibilité du code.

répertoires ainsi que leurs profondeur est sans limite. Cependant la compréhension du code doit rester claire.

```
// main.c
#include "json.h"
```

```
gcc -o main.o -c main.c -Ilib/json
```

FIGURE 3.5 – Exemple de commande à ne pas utiliser

Dans la figure ci contre nous ne somme pas capable au premier regard de savoir ou ce trouver le fichier json.h, et ainsi il deviens plus compliqué de retravailler le code. De plus dans le cas ou vous utilisez un IDE, et que vous souhaitez repasser sur un outil libre par la suite vous devrez reprendre tous vos code sources pour replacer tous les path aux bon endroits.

Au contraire utiliser des path complets permet d'avoir un code beaucoup plus lisible bien que cela demande un peut plus de reflexion au debut du projet pour savoir comment seront organisés les fichiers.

```
// main.c
#include "lib/json/json.h"
```

```
gcc -o main.o -c main.c
```

FIGURE 3.6 – Exemple de ce qui est préconisé

3.1.4 outils

Dans certains cas nous voulons garder les outils que nous utilisons pour une version particulière de nos projet, cela est compréhensible et parfois nécessaire pour des logiciels qui sont certifiés.

Bien qu'il puisse être tentant de placer les outils. sources dans les projet à proprement parler, ceci est à éviter. Cela va alourdir le projet et le rendre plus complexe à manipuler tout en augmentant de façon artificiel son volume. Imaginez que vous utilisiez `GCC` 4.10 pour la version 1 de votre projet et que pour la version deux vous décidez d'utiliser la version 7.5. Vous devrez traîner deux version de `GCC` à chaque clone, même si vous n'en utilisez qu'une.

Nous vous préconisons dans ce cas de créer un projet annexe pour chaque outils (fork/clone) et dans votre projet principal n'utiliser qu'un lien vers l'outil en question.

3.2 Fichiers

3.2.1 Source.c

Un ficher source est un fichier d'extension `*.c` ou `*.cpp` ; il doit forcément être accompagné d'un fichier header avec l'extension `*.h` (à l'exception du main qui peut s'en dispenser).

Un fichier source doit inclure tous les fichiers (`*.h`) qui lui sont utiles, ainsi que son *.h personnel.

Il est de bon aloi de limiter la taille d'un fichier source. Nous préférerons avoir cinquante fichiers de 200 lignes qu'un fichier de 10 000 lignes. (Avec l'utilisation d'un makefile bien fait cela permet de réduire les durées de compilation).

Un fichier source peut avoir plusieurs headers.

3.2.2 Header.h

Tous les fichiers headers doivent avoir des barrières d'inclusions (protection contre les inclusions récursives).

Les barrières d'inclusions portent le nom du fichier, sans espaces, en MAJUSCULE, les points sont remplacés par des « _ » et elles commencent et se terminent par un double underscore « __ ».

Un fichier d'en-tête doit avoir le même nom que le fichier source afférant, à l'exception de l'extension en *.h. Si un fichier source à plusieurs fichiers d'entête, ils doivent clairement lui faire référence de par leur dénomination.

```
#ifndef __FUNCTION_H__
#define __FUNCTION_H__

...

#endif
```

FIGURE 3.7 – Fichier function.h

Un fichier d'en-tête ne doit pas contenir de code (excepté des macros, plus de détails si après).

3.3 Commentaires

3.3.1 Source

Deux sortes de commentaires sont à différencier : ceux placés dans les codes sources (`*.c*.cpp`) doivent obligatoirement être mono-lignes, et ne servent qu'au développeur voulant reprendre votre code. Ce type de commentaire n'a pour visé que l'explication de zones locales du code ou de spécificités de conception (algorithmeoptimisation...). Les explications quand à l'utilisation d'une fonction ne se font pas dans le code source mais dans le header.

```
int i = 0; // compteur de boucle
int j = 0; // compteur de boucle
char outFileAddr[20] = { 0 }; // adresse du fichier de sortie
```

FIGURE 3.8 – Déclaration des variables

3.3.2 Header

Dans le header, nous trouverons des commentaires à destination du programmeur utilisant la fonction, nous aurons donc les explications sur les paramètres des fonctions, leurs retours ainsi que les bugs éventuels.

Nous utilisons le format Doxygen pour tous les commentaires situés dans les headers.

```
////////////////////////////////////////////////////////////////////////////////
/// \file mallocND.h
/// \author ox223252
/// \date 2016-06
/// \version 0.0
////////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
/// \fn int malloc2D ( const uint8_t sizeOfType , const uint64_t a,
///  const uint64_t b, void ***ptr );
/// \param[in] sizeOfType: nombre d'octet du type (sizeof(type))
/// \param[in] a: première dimension du tableau
/// \param[in] b: seconde dimension du tableau
/// \param[in/out] ptr: pointeur NULL sur le tableau de sortie
/// \return 0 succès
/// \return 1 pointeur non NULL
/// \return 2 valeur non géré pour sizeoftype
/// \brief alloue la mémoire pour un tableau deux dimensions ptr[a][b] de type
///  paramétré par le "sizeOfType"
/// \author ox223252
/// \bug pas de bug connus
////////////////////////////////////////////////////////////////////////////////
int malloc2D ( const uint8_t sizeOfType , const uint64_t a, const uint64_t b,
    void ***ptr );
```

FIGURE 3.9 – Fichier mallocND.h

Les commentaires présents dans les headers doivent être taillés à 80 digits par ligne.

3.4 Define & macros

Attention les définition et macros sont à éviter cf section sur le Macros.

3.4.1 Typographie

Les définitions de constantes sont en MAJUSCULES. Les définitions de macros sont en MAJUSCULES.

```
#define FONCTION_DEBUG_TRACE(...) {printf(__VA_ARGS__);}
#define ARRAYSIZE 10
```

FIGURE 3.10 – Typographie de définitions

3.4.2 Portée

Les définitions et macros utiles que dans un seul fichier source doivent se trouver dans le dit fichier source.

```
#define FONCTION_DEBUG_TRACE(...) {printf(__VA_ARGS__);}
```

FIGURE 3.11 – fonction.c

Les définitions et macro relatives à un fichier source mais pouvant être appelées ailleurs dans le code doivent se trouver dans le fichier header.

```
#define ARRAYSIZE 10
```

FIGURE 3.12 – fonction.h

Les macros ou définitions pouvant impacter tout le code peuvent être placées dans un fichier header qui n'aura que cette utilité.

```
#define MAX(x,y) ((x < y)?y:x)
#define PI 3.14159
```

FIGURE 3.13 – Fichier define.h

3.4.3 Typage

Comme exprimé en début de paragraphe précédent, nous recommandons très vivement de ne pas utiliser de `define`.

Pour les fonctions macroscopiques, elles peuvent être remplacées par des fonctions `inline`. Cela permet la vérification des types et une plus grande sécurité quant à des effets de bord indésirables :

```
// deprecated
#define CARRE(x) ((x)*(x))
...
b = CARRE(a++); // b = ((a++)*(a++))
....
```

FIGURE 3.14 – Contre exemple d'utilisation du define pour des macros

```
inline int carre(x)
{
    return x * x;
}
...
b = carre(a++); // b = a*a; a++;
....
```

FIGURE 3.15 – Exemple d'utilisation du inline

De même que pour les macros, les définitions de constantes sont à supprimer au maximum au profit d'énumérations ou de variables `static const`. Cela permet au compilateur de faire les vérifications de typage.

De plus, tout `enum` doit commencer par une racine identifiant la classe afin d'éviter les doublons.

```
#define ERROR_ADDR 0x01
#define ERROR_FLAG 0x02
#define TEMPO 12
...
```

FIGURE 3.16 – Contre exemple d'utilisation du define pour des constantes

```
typedef enum
{
    ERROR_CODE_OK, ///< retour normal
    ERROR_CODE_ADDR, ///< retour de fonction en cas d'adresse incorrecte
    ERROR_CODE_FLAG ///< retour de fonction en cas de flag incorrecte
}
ERROR_CODE;
const uint8_t TEMPO = 12; ///< temporisation après mise à HIGH du pin 13
```

FIGURE 3.17 – Exemple d'enum

3.5 Variables

3.5.1 Typographie

Les noms de variables doivent respecter la norme `camelCase`, soit s'écrire en supprimant les espaces et en mettant la première lettre de chaque mot en MAJUSCULE excepté la première lettre du premier mot.

```
int compteurDeBoucle = 0;
```

Les variables doivent avoir un nom explicite.

Les variables doivent avoir un nom de plus de deux lettres (excepté pour les compteurs de boucles locales. Elles doivent néanmoins être commentées lors de leur déclaration).

Même si le nom de la variable est explicite, il est très fortement conseillé de déclarer chaque variable sur une ligne séparée en début de programme et d'expliquer son utilité.

```
int i = 0; // compteur de boucle
int j = 0; // compteur de boucle
char outFileAddr[20]; // adresse du fichier de sortie
```

FIGURE 3.18 – Déclaration des variables

Toutes les variables globales doivent être définies et initialisées dans les fichiers sources (`*.c`) et doivent être préfixe par un `g_`. Elles doivent être déclarées dans les headers en `extern`, cela évitera les inclusions multiples et donc les multiples définitions.

```
char *g_outputFolder = NULL;
```

FIGURE 3.19 – Fichier fonction.c

```
#ifndef __FONCTION_H__
#define __FONCTION_H__
extern char *g_outputFolder; ///< nom du fichier de sortie
#endif
```

FIGURE 3.20 – Fichier fonction.h

Si vous avez des variables « semi-globales » (globales pour un seul fichier source), vous pouvez les déclarer dans le fichier source, mais lors de leur nommage ajoutez leur le préfixe : `_<fileName>_varName`.

```
char *_fonctions_outputFolderPath; ///< nom du dossier de sortie
```

FIGURE 3.21 – Fichier fonctions.c

3.5.2 Déclaration

Les variables doivent être déclarées de façon à avoir la portée la plus courte possible, ainsi elles devront être déclarées juste avant leur utilisation.

```
if ( ... )
{
    uint8_t variable = 0; // variable qui n'aura comme durée de vie que ce bloc de code
}
```

FIGURE 3.22 – Exemples des types disponibles

3.5.3 Initialisation

Une variable, de même qu'un tableau mémoire, doit toujours être initialisé ~~en début de programme~~ lors de sa déclaration.

```
char *string = NULL; // pointeur initialisé à null
char *string2[20] = { 0 }; // zone mémoire initialisée à 0
uint32_t veleur = 0; // variable initialisée à zéro
```

FIGURE 3.23 – Initialisation

Même si par défaut beaucoup de compilateurs initialisent la mémoire à 0, il arrive que lors d'appels multiples à une fonction, une même variable se retrouve sur sa précédente zone mémoire et récupère sa précédente valeur. Il est donc nécessaire de fixer la valeur d'initialisation.

3.5.4 Typage

Pour chaque environnement définissant ses variables à sa guise, nous recommandons l'utilisation de types explicites définis dans la bibliothèque `stdint.h`.

```
#include <stdint.h>

int8_t ///< entier signé sur 8 bits
uint8_t ///< entier non-signé sur 8 bits
int16_t ///< entier signé sur 16 bits
uint16_t ///< entier non-signé sur 16 bits
int32_t ///< entier signé sur 32 bits
uint32_t ///< entier non-signé sur 32 bits
int64_t ///< entier signé sur 64 bits
uint64_t ///< entier non-signé sur 64 bits
```

FIGURE 3.24 – Exemples des types disponibles

3.5.5 Portée

Les variables doivent être déclarées de façon à avoir la portée la plus courte possible. Ainsi, pour cette structure :

```
uint8_t a = 0;
uint8_t tmp = 0;
if( ... )
{
tmp = ...;
}
```

On préférera utiliser :

```
uint8_t a = 0;
if( ... )
{
uint8_t tmp = ...;
}
```

Les variables globales sont à éviter au maximum, si vous pouvez, les remplacer par des variables locales ou « semi-globales » à défaut.

3.5.6 Libération

Pour les pointeurs, une fois la libération de la mémoire effectuée, il est nécessaire de remettre le pointeur à `NULL` manuellement. Tous les compilateurs ne le font pas automatiquement.

```
free ( ptr );
ptr = NULL;
```

FIGURE 3.25 – Libération d'un pointeur

3.6 Fonctions

3.6.1 Typographie

Les noms de fonctions doivent respecter la norme `camelCase`, soit s'écrire en supprimant les espaces et en mettant la première lettre de chaque mot en MAJUSCULE excepté la première lettre du premier mot.

```
int fonctionDeTest ( void )
{
    ...
```

Si une ou plusieurs fonction appartiennent à un `objet`/`class` alors on pourra préfixer le nom de la fonction par le nom de l'objet avec un underscore pour séparer les deux `strings`.

```
uint32_t actionManager_stepGetNext ( void );
uint32_t actionManager_stepGetAction ( void );
uint32_t actionManager_stepGeturrentAction ( void );
```

Les fonctions doivent avoir un nom explicite.

3.6.2 Paramètres

Les paramètres des fonctions doivent être au maximum passés en `const` et `restrict` quand c'est possible.

```
int function ( const int * restrict a, int * restrict const b )
{
    ...
```

3.6.3 Retour

Utiliser le `return` pour un retour d'état (réussit, échoue avec un code d'erreur explicite), le retour 0 doit être le retour normal d'une fonction si tout se passe bien, seul les fonctions renvoyant une valeur `max`/`min`/`countNbOfThing` peuvent déroger à cette règle.

Il est donc de bon aloi de définir tous les retours possibles dans un `enum`. Dans le cas d'une utilisation de `errno` avec les codes erreurs standard, il est acceptable de ne pas définir tous les codes erreurs, mais simplement retourner le numéro de ligne en cas d'echec.

```
typedef enum
{
    ERROR_CODE_OK = 0,
    ERROR_CODE_ADDR,
    ERROR_CODE_FLAG
}
ERROR_CODE;
ERROR_CODE function ( const char *path, const int64_t param1 );
```

FIGURE 3.26 – Fichier fonctions.h

```
errno = EINVAL;
return ( __LINE__ );
```

FIGURE 3.27 – Fichier fonctions.c

3.6.4 portée

Les fonctions dont la portée est limitée au fichier lui-même doivent être définies en `static`, et ne doivent pas être déclarées dans le fichier header correspondant.

3.7 Général

3.7.1 Parenthèses

Dans les conditions, affectations et autres, on ne doit pas utiliser les règles mathématiques de priorité implicite mais définir clairement avec des parenthèses l'ordre d'exécution. Car selon les compilateurs, l'interprétation peut se faire différemment.

~~Seuls les opérations logiques de conditions peuvent utiliser les priorités implicites.~~

Nous placerons un espace autour de chaque parenthèse.

```
a = b >> 4 + c;
a = ( b >> 4 ) + c;
a = b >> ( 4 + c );
```

FIGURE 3.28 – Placement des parenthèses sur oppérations

```
if ( ( a == 0 ) ||
    ( ( b > 10 ) &&
    ( b < 23 ) ) )
{
    ...
```

FIGURE 3.29 – Placement des parenthèses sur conditions

3.7.2 Accolades

Les accolades étant un grand sujet de dissension au sein des équipes de programmeurs, ici nous imposons que les accolades ouvrantes et fermantes doivent se trouver seules sur leur ligne (commentaires autorisés).

De plus toutes instructions conditionnelles ou de boucles doivent être suivies d'un bloc entre accolades même si cette instruction n'est suivi que d'une seule ligne.

```
int main ( int argc, char **argv )
{
    if ( argc < 2 )
    { // commentaire autorisé si besoin
        printf ( "hey\n" );
    }
    else
    { // commentaire autorisé si besoin
        printf ( "ho\n" );
    }
    return ( 0 );
}
```

FIGURE 3.30 – Placement des accolades

Ça évide le risque d'ajouter une instruction à la suite de l'indentation et de ne pas voir que cette instruction sera systématique.

3.7.3 Indentation

L'indentation doit se faire avec des tabulations de 4 éléments de largeur.

Nous n'utilisons pas les espaces pour indenter le code, à l'exception des headers (plus de détails ci après).

3.7.4 Sizeof

Utilisez de préférence `sizeof ( varName )` à `sizeof ( type )`.

L'utilisation de `sizeof ( varName )` quand on cherche à connaître la taille d'un variable va permettre de ne pas dépendre de la définition de la variable (utile dans le cas pour le type de la variable change).

3.7.5 structures

Les structures utilisées dans un seul et unique fichier source doivent être définies dans ce même fichier source.

Les structures afférentes à un fichier source mais utilisées ailleurs, doivent être définies dans le fichier header correspondant (`*.h`) et documenté au format Doxygen.

```
#ifndef FONCTION_H
#define FONCTION_H

/// \typedef Point
/// \brief structure contenant les coordonnées d'un point
typedef struc _point
{
    int32_t x; ///< coordonnée X (largeur)
    int32_t y; ///< coordonnée Y (hauteur)
}
Point;

#endif
```

FIGURE 3.31 – Fichier fonction.h

3.8 Aide & usage

Vous devez quand vous écrivez un programme, toujours ajouter une notice d'aide quant à l'utilisation du dit programme. Cette notice sera aussi affichée en cas d'appel incorrect au programme.

L'aide doit être affichée dans tous les cas ou, le programme est appelé avec le mauvais nombre d'arguments, une commande invalide ou la commande d'aide.

Un programme nécessitant au minimum un argument doit dans le cas d'un appel sans argument afficher l'aide, ou un message indiquant comment avoir l'aide.

```
> ./monProgramme
use ./monProgramme -h
```

FIGURE 3.32 – Exemple de retour en cas d'appel invalide à un programme

Cet affichage doit indiquer :
— le nom du programme,
— l'utilité du programme,
— le nombre d'arguments
— à quoi correspond chaque argument
— si un argument est optionnel ou non
— toute autre information relative aux arguments

Le nom du programme ne doit pas être codé en dur dans le logiciel mais récupéré de la ligne de commande.

```
int main ( int argc, char * argv[] )
{
    argv[ 0 ] // nom du programme
}
```

FIGURE 3.33 – Récupération du nom du programme

Attention : ne pas utiliser les symboles `[ ] < >` ils ont une signification particulière décrite plus loin, de même les arguments de la ligne de commande ne doivent pas être séparés pas un virgule ou des points. Laissez la ligne d'exemple le plus proche possible de ce que l'utilisateur devrait saisir dans son terminal.

```
> ./monProgramme -h
monProgramme lit des données les traites et écrit le retour dans un fichier de sortie.
usage: monProgramme inputFile outputFile
    inputFile : fichier de données utilisé en entrée
    outputFile : fichier de sortie contenant les données traitées
```

FIGURE 3.34 – Exemple d'aide avec arguments obligatoires

Beaucoup de programmes utilisent des paramètres optionnels. Si un paramètre est optionnel il doit être encadré par des crochet `[arg]`, vous pouvez avoir deux types de notation : la notation courte `[-a]`, et la notation longue `[--arg]`. un argument optionnel peut avoir son propre paramètre optionnel :

```
> ./monProgramme -h
monProgramme lit des données les traites et écrit le retour dans un fichier de sortie.
usage: monProgramme inputFile outputFile [-a value ... ] [-v [-d]] [-h]
    inputFile : fichier de données utilisé en entré
    outputFile : fichier de sortie contenant les données traitées
    -a | --arg valeur ... : change la donnée d'entée (max 100 valeurs)
    -v | --verbose : affiche des informations durant le traitement
    -d : affiche plus de détail
    -h | --help : affiche cette page
```

FIGURE 3.35 – Exemple d'aide avec arguments optionnels

Si vous observez bien le mode `debug -d` n'est disponible que si et seulement si le mode `verbose` est activé `-v`.

Si vous observez les paramètres, pour `-a` vous pouvez saisir plusieurs valeurs (...), il est donc obligatoire de vérifier le nombre et le contenu des arguments, pour vous assurer de leurs validités.

Les symboles `< >` peuvent être utilisés pour faire des redirections unix, ainsi :

```
monProgramme < inFile > outFile
```

Peut être interprété comme suit :

```
./monProgramme < input_file > output_file
```

Cette commande utilise le contenu de `input_file` comme entrée standard plutôt que le `stdin` et utilise le fichier `output_file` à la place de `stdout`. Pour cette raison les opérateurs `< >` sont réservés aux redirections.

3.9 Condition

3.9.1 switch/case

Un `swich/case` sur une valeur autre qu'un `enum` doit avoir une action `default`.

```
switch ( valeur )
{
    ...
    dafault:
    { // action par defaut
        break;
    }
}
```

FIGURE 3.36 – Conditions multiples

Un `swich/case` sur un `enum` doit explicitement traiter tous les valeurs possibles de l'`enum` ou avoir une action `default`.

Cela permet d'être plus souple quand à l'évolution d'une fonction

```
// fichier header
typedef enum
{
    ERROR_CODE_OK,
    ERROR_CODE_ADDR,
    ERROR_CODE_FLAG
}
ERROR_CODE;

ERROR_CODE function ( const char *path, const int64_t param1 );

// fichier source
switch ( function ( "path", param1 ) )
{
    case ERROR_CODE_OK:
    { // tout va bien dans le meilleur des mondes
        break;
    }
    case ERROR_CODE_ADDR:
    case ERROR_CODE_FLAG:
    dafault:
    { // error management
        break;
    }
}
```

FIGURE 3.37 – Conditions multiples

3.9.2 Instructions conditionnelles

Une instruction conditionnelle à multiples éléments ne doit avoir qu'un élément de condition par ligne, permettant ainsi une meilleure lisibilité du code.

```
if ( ( a == 0 ) ||
    ( ( b > 10 ) &&
    ( b < 23 ) ) )
{
...
```

FIGURE 3.38 – Conditions multiples

3.10 Documentation

Les commentaires placés dans les headers doivent être au format Doxygen. Ils doivent être écrits en Anglais avec le plus d'éléments d'explications possibles. Quant à l'utilisation du code, vous pouvez de plus en faire une traduction dans votre langue maternelle – si votre langue maternelle n'est pas l'anglais – comme dans l'exemple suivant :

```
#ifdef English_dox
    /// Defined by the C++ standard library.
#endif
#ifdef Spanish_dox
    /// Definido por la biblioteca estándar C++.
#endif
#ifdef French_dox
    /// Définition pour la bibliotheque C++.
#endif
```

FIGURE 3.39 – Doxygen multi-langues

Pour générer le Doxygen dans une seule langue, vous devrez dans le fichier Doxyfile définir quelle langue doit être utilisée pour la génération :

```
OUTPUT_LANGUAGE        = French
OUTPUT_DIRECTORY       = French
ENABLE_PREPROCESSING   = YES
MACRO_EXPANSION        = YES
PREDEFINED             = French_dox
```

FIGURE 3.40 – Doxygen Doxyfile

3.10.1 Multi-langue

Comme on peut le voir ici, la documentation peut devenir très volumineuse, il est donc recommandé de séparer la documentation dans des fichiers headers différents pour chaque langues.

```
#ifndef __MALLOCNC_FR_H__
#define __MALLOCNC_FR_H__

#ifndef French_dox
#define French_dox
#endif

#ifdef French_dox
////////////////////////////////////////////////////////////////////////////
/// \file mallocND.fr.h
/// \author ox223252
/// \date 2016-06
/// \version 0.0
////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdint.h>

////////////////////////////////////////////////////////////////////////////
/// \fn int malloc2D ( const uint8_t sizeOfType , const uint64_t a,
///  const uint64_t b, void ***ptr );
/// \param[in] sizeOfType: nombre d'octet du type (sizeof(type))
/// \param[in] a : première dimension du tableau
/// \param[in] b : seconde dimension du tableau
/// \param[in/out] ptr : pointeur NULL sur le tableau de sortie
/// \return 0 success
/// \return 1 pointeur non NULL
/// \return 2 valeur non gérée pour sizeoftype
/// \brief alloue la mémoire pour un tableau deux dimensions ptr[a][b] de type
///  paramétré par le "sizeOfType"
/// \author ox223252
/// \bug pas de bug connus
////////////////////////////////////////////////////////////////////////////
int malloc2D ( const uint8_t sizeOfType , const uint64_t a, const uint64_t b,
    void ***ptr );
#endif
#endif
```

FIGURE 3.41 – Fichier mallocND.fr.h

```
#ifndef __MALLOCNC_EN_H__
#define __MALLOCNC_EN_H__

#ifndef English_dox
#define English_dox
#endif

#ifdef English_dox
////////////////////////////////////////////////////////////////////////////////
/// \file mallocND.en.h
/// \author ox223252
/// \date 2016-06
/// \version 0.0
/// \copyright ox223252, 2016-06
////////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdint.h>

////////////////////////////////////////////////////////////////////////////////
/// \fn int malloc2D ( const uint8_t sizeOfType , const uint64_t a,
///  const uint64_t b, void ***ptr );
/// \param[ in ] sizeOfType: type length (sizeof(type))
/// \param[ in ] a: first dimention of ptr array
/// \param[ in ] b: second dimention of ptr array
/// \param[ in/out ] ptr: NULL pointer on the out array
/// \return 0: success
///  1: non NULL pointer
///  2: non valid value for sizeoftype
/// \brief allow memory for a 2D array ptr[ a ][ b ]
/// \author ox223252
/// \bug non bugs known
////////////////////////////////////////////////////////////////////////////////
int malloc2D ( const uint8_t sizeOfType , const uint64_t a, const uint64_t b,
    void ***ptr );
#endif
#endif
```

FIGURE 3.42 – Fichier mallocND.en.h

3.10.2 Mots clés

`\attention`	débute un paragraphe "attention"
`\author`	pour donner le nom de l'auteur.
`\brief`	pour donner une description courte.
`\code{type}` ... `\endcode`	place un bloc de code.
`\copyright`	pour indiquer un copyright.
`\def`	pour documenter un `#define`.
`\deprecated`	pour spécifier qu'une fonction / variable...n'est plus utilisée.
`\enum`	pour documenter un type énuméré.
`\example`	pour écrire définir une fonction.
`\file`	pour documenter un fichier.
`\fixme`	pour indiquer un code défectueux, « à réparer »
`\fn`	pour documenter une fonction.
`\li`	pour faire une puce.
`\note`	paragraphe de notes.
`\par`	[title] paragraphe.
`\param`	pour documenter un paramètre de fonction/méthode.
`\remark`	paragraphe de texte indenté.
`\return`	pour documenter les valeurs de retour d'une méthode/fonction.
`\see`	pour renvoyer le lecteur vers quelque chose (une fonction, une classe, un fichier...).
`\since`	pour faire une note de version (ex : disponible depuis...).
`\struct`	pour documenter une structure C.
`\todo`	pour indiquer une opération restant « à faire ».
`\typedef`	pour documenter la définition d'un type.
`\union`	pour documenter une union C.
`\var`	pour documenter une variable / un typedef / un énuméré.
`\version`	pour donner le numéro de version.
`\warning`	pour attirer l'attention.

FIGURE 3.43 – Commandes Doxygen usuelles

3.10.3 Formules Mathématiques

Si dans une fonction vous utilisez des formules mathématiques devant être notées, pour informer l'utilisateur, ou pour des raisons de certifications, utilisez les formules LaTeX, elles sont comprises par Doxygen :

```
The distance between \f$(x_1,y_1)\f$ and \f$(x_2,y_2)\f$ is
    \f$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\f$.
```

The distance between (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

FIGURE 3.44 – Exemples de formules mathématiques (1)

```
\f[
    |I_2|=\left| \int_{0}^T \psi(t)
            \left\{
                u(a,t)-
                \int_{\gamma(t)}^a
                \frac{d\theta}{k(\theta,t)}
                \int_{a}^\theta c(\xi)u_t(\xi,t)\,d\xi
            \right\} dt
        \right|
\f]
```

$$|I_2| = \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^{a} \frac{d\theta}{k(\theta,t)} \int_a^{\theta} c(\xi) u_t(\xi,t)\, d\xi \right\} dt \right|$$

FIGURE 3.45 – Exemples de formules mathématiques (2)

`$ formule\ mathématique $`	->	$formule\ mathmatique$
`\int_{indice}^{exposant}`	->	$\int_{indice}^{exposant}$
`\frac{diviseur}{dividende}`	->	$\frac{diviseur}{dividende}$
`\left| valeur \right|`	->	$\left\| valeur \right\|$
`{ valeur }`	->	$valeur$
`\psi(a)`	->	$\psi(a)$
`\theta(b)`	->	$\theta(b)$
`\gamma(c)`	->	$\gamma(c)$
`\xi(d)`	->	$\xi(d)$
`\alpha(e)`	->	$\alpha(e)$
`\beta(f)`	->	$\beta(f)$

FIGURE 3.46 – Mots clés

Vous pouvez définir `USE_MATHJAX` pour avoir un meilleur rendu des formules :

```
USE_MATHJAX            = YES
```

FIGURE 3.47 – Doxygen Doxyfile

JS

4.1 Commentaires

Deux sortes de commentaires sont à différencier, les commentaires de développement doivent obligatoirement être mono-lignes, et ne servent qu'au programmeur voulant reprendre votre code, ce type de commentaire n'a pour visé que l'explication de zones locales de code ou de spécificités de conception (algorithmeoptimisation...), ils n'ont pas besoin de suivre une syntaxe spécifique.

```
var i = 0; // compteur de boucle
var j = 0; // compteur de boucle
var outFileAddr = ""; // adresse du fichier de sortie
```

FIGURE 4.1 – Déclaration des variables

Les explications quand à l'utilisation d'une fonction se font en suivant le format Doxygen.

```
////////////////////////////////////////////////////////////////////////////////
/// \file monModule.js
/// \author ox223252
/// \date 2016-06
/// \version 0.0
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
/// \fn myFunction ( path, file, folder, callback );
/// \param[ in ] path: chemin du repertoir de travail
/// \param[ in ] file: fichier de donnée en entré
/// \param[ in ] folder: dossier de sortie
/// \param[ in ] callback: fonction de callback
/// \return 0 succes
/// \return 1 path invalide
/// \return 2 file invalide
/// \return 3 folder non vide
/// \brief fait quelque chose d'utile.
/// \author ox223252
/// \bug pas de bug connus
////////////////////////////////////////////////////////////////////////////////
function myFunction ( path, file, folder, callback )
{
    ...
}
```

FIGURE 4.2 – Fichier monModule.js

4.2 Les variables

4.2.1 Typographie

Les noms de variables doivent respecter la norme `camelCase`, soit s'écrire en supprimant les espaces et en mettant la première lettre de chaque mot en MAJUSCULE excepté la première lettre du premier mot.

```
var compteurDeBoucle = 0;
```

Les variables doivent avoir un nom explicite.

Les variables doivent avoir un nom de plus de deux lettres (excepté pour les compteurs de boucles, ils doivent néanmoins être commentés lors de leur

déclaration).

Même si le nom de la variable est explicite il est très fortement conseillé de déclarer chaque variable sur une ligne séparée en début de programme et d'expliquer son utilité.

```
var i = 0; // compteur de boucle
var j = 0; // compteur de boucle
var outFileAddr = ""; // adresse du fichier de sortie
```

FIGURE 4.3 – Déclaration des variables

4.2.2 Globales

L'usage de variables globales doit être évitée au maximum : On distingue comme variable l'ensemble des types de données y compris les fonctions. Les variables déclarées globales ont une portée (scope) sur toute l'application et peuvent réécrire ou se faire réécrire par d'autres parties du code.

Eviter leur usage améliore la maintenance et réduit le couplage entre les différentes parties d'une application.

4.2.3 Locales

Les variables locales doivent être déclarées à l'aide du mot clé `var`. Cela empêche la déclaration d'une variable locale dans l'environnement global. Cette règle découle de la précédente.

Cela permet une meilleur maîtrise de la portée des variables.

4.2.4 Parsing

La base doit toujours être spécifiée lors de l'usage de parseInt. `parseInt()` accepte comme second argument la base de conversion. Définir cette base permet d'éviter des erreurs, en particulier si la chaîne commence par 0 ce qui aura pour effet une conversion en base 8 (octale).

4.3 strict & evals

4.3.1 mode strict

Le mode `strict` doit être utilisé.

Depuis la version 5 d'Ecmascript (Javascript 1.8.5), il est possible d'utiliser la directive `use strict` pour forcer l'exécution du code en mode strict. Ce mode permet de remonter des erreurs qui sont par défaut silencieuses.

Grâce à ce mode, on évite par exemple l'oubli du mot clé `var`. Du fait que le javascript est un langage assez permissif, le mode strict augmente la sécurité du code.

4.3.2 evals

L'usage de `eval()` doit être évité. `eval()` permet de parser et d'évaluer une expression à travers une nouvelle instance de l'interpréteur javascript. Cette fonction est coûteuse en terme de performances mais peut surtout devenir une faille de sécurité importante.

4.4 Les modules

Les fichiers et / ou principales fonctionnalités d'un programme devraient être enrobées dans un module Cette règle rejoint la règle à propos des variables globales (cf ref 7.1). La portée d'un module étant limitée, on obtient une plus grande maîtrise du code et on évite aussi la propagation de variables à d'autres fonctions. On peut utiliser plusieurs méthodes pour y parvenir : AMD , CommonJS, ES6, objet en notation littérale, module pattern (IIFE).

```
// Cette exemple montre l'utilisation d'une IIFE pour réaliser un module
var monModule = (function ( o )
{
    function methodePrivee ( )
    {
        o.doSomthing();
    }

    function methodePrivee2 ( )
    {
        console.log ( "hello" );
    }

    return
    {
        methodePublique: function ( )
        {
            methodePrivee ( );
        }
    };
    // Exemple d'import d'un objet
})( obj );
monModule.methodePublique ( );
```

FIGURE 4.4 – Modules

4.5 Règles de syntaxe

4.5.1 Parenthèses

Dans les conditions, affectations et autres, on ne doit pas utiliser les règles mathématiques de priorité implicite mais définir clairement avec des parenthèses l'ordre d'exécution. Car selon les compilateurs, l'interprétation peut se faire différemment.

Seuls les opérations logiques de conditions peuvent utiliser les priorités implicites.

Nous placerons un espace autour de chaque parenthèse.

```
a = b >> 4 + c;
a = ( b >> 4 ) + c;
a = b >> ( 4 + c );
```

FIGURE 4.5 – placement des parenthèses sur oppérations

```
if ( ( a == 0 ) ||
    ( b > 10 ) &&
    ( b < 23 ) )
{
    ...
```

FIGURE 4.6 – placement des parenthèses sur conditions

4.5.2 Accolades

Toutes les instructions conditionnelles ou de boucles doivent toujours utiliser des accolades, même si cette instruction n'est suivie que d'une seule ligne. Délimiter une instruction par un bloc (`{}`) permet de clarifier l'instruction à exécuter pour le développeur ainsi que pour l'interpréteur.

Les accolades ouvrantes et fermantes doivent se trouver seules sur leur ligne (commentaires autorisés).

```
if ( value < 2 )
{ // commentaire autorisé si besoin
    console.log ( "hey" );
}
else
{ // commentaire autorisé si besoin
    console.log ( "ho" );
}
```

FIGURE 4.7 – Placement des accolades

4.5.3 Indentation

L'indentation doit se faire avec des tabulations de 8 éléments de largeur. Nous n'utilisons pas les espaces pour indenter le code.

4.5.4 Notation littérale

La notation littérale doit être utilisée pour créer des objets, tableaux, expressions régulières et primitives. Elle permet d'éviter de réécrire un objet original en cas d'omission de l'opérateur `new`. Vous sécuriserez ainsi le code.

```
// ecrire
var monObjet = {} ;
var monTableau = [] ;
// Plutot que
var monObjet = new Object() ;
var monTableau = new Array() ;
```

FIGURE 4.8 – New objects

4.5.5 l'égalité

Par défaut, l'égalité stricte doit être utilisée. La stricte égalité (`===`) permet de vérifier à la fois la valeur et le type. Dans le cas de la simple égalité (`==`), l'interpréteur essayera de faire correspondre le type des éléments comparés (coercition).

```
var a1 = ('1' == 1); // true
var a2 = ('1' === 1); // false
var a3 = ('true' == true); // true
var a4 = ('true' === true); // false
```

FIGURE 4.9 – Égalité stricte

Ainsi vous aurez une plus grande maîtrise lors des comparaisons et éviterez des résultats non souhaités suite à une conversion de type automatique.

4.5.6 Point virgule

Une expression doit toujours se terminer par un point-virgule. Même si la syntaxe ne l'oblige pas, cette pratique limite le risque d'erreurs, permet la minification et une meilleur analyse de la qualité du code.

4.6 Navigateur

Règles appliquées au javascript coté navigateur.

Les scripts javascript doivent être placés dans des fichiers externes. Le javascript dans des fichiers séparés permet de correctement distinguer le code source du corps de page HTML, et ainsi avoir une meilleur maintenabilité.

Les scripts javascript doivent être appelés de préférence en fin de page. L'interprétation d'un fichier HTML par le navigateur est linéaire et bloquante, de cet fait, appeler les fichiers sources en bas de page rend l'affichage initial de la page plus rapide et permet d'éviter l'appel à des événements de type onReady.

Dans le cas d'un code qui doit interagir avec la page avant l'affichage complet du DOM, on pourra placer ce script dans l'en-tête.

Ceci améliore la lisibilité et augmentation des performances.

4.7 Performances

4.7.1 Selecteurs

Les sélecteurs doivent être mis en cache lors d'un usage répété. L'interaction avec le DOM est lente. Lors de l'usage répété d'un sélecteur, il convient de le mettre en cache pour éviter de retraverser le DOM.

```
// Mauvais
for (var i = 0; i < 10000; i++)
{
    $('#container').html(i);
}
// Bon
var monElement = $('#container')
for (var i = 0; i < 10000; i++)
{
    monElement.html(i);
}
//Meilleur ( usage d'un sélecteur natif )
var monElement = document.getElementById(container)
for (var i = 0; i < 10000; i++)
{
    monElement.innerHTML = i;
}
```

FIGURE 4.10 – Selecteur

Vous augmenterez ainsi les performances.

4.7.2 Bibliotheques

Lorsque cela est possible, les fonctions natives du DOM doivent être utilisées. Les navigateurs ont atteint une maturité qui permet de se passer de bibliothèques spécialisées dans la plupart des cas courants (sélection d'éléments, interaction avec le DOM, appels AJAX). C'est particulièrement vrai pour la sélection d'éléments du DOM. L'usage d'une bibliothèque externe telle que jQuery uniquement dans ce but est inutile aujourd'hui.

Ainsi vous pourrez limiter la complexité du code et la dépendance à des bibliothèques externes.

4.7.3 Document.write()

La fonction `document.write()` ne devrait pas être utilisée. Cette fonction modifie le DOM pendant la lecture de la page par le navigateur. Cela peut porter à confusion et potentiellement bloquer le rendu de la page. Il est préférable d'utiliser des fonctions de manipulation du DOM une fois la page totalement chargée. Son usage doit être limité à des cas particuliers.