Guide de style pour Ella

28 novembre 2010

Table des matières

1 Fichier

1.1 Longueur des lignes

Les lignes auront une longueur maximale de 80 caractères afin d'être lisible sur tous type de terminaux.

1.2 En-tête des fichiers

Chaque fichier (header et source) commencera par un bloc de commentaire qui contiendra les champs suivant :

- le copyright et les termes de redistributions (la licence) ;
- nom du fichier ;
- description du fichier (= module) ;
- nom, prénom et adresse mail des auteurs (un par ligne).

Ces informations seront présentées de la façon suivante.

```
/* Copyright (C) 2010 Laperche Sylvain, Mouelet Leandre
 *
 * This file is part of Maze Project.
 *
 * Maze Project is free software : you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * Maze Project is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY ; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with Maze Project.  If not, see <http ://www.gnu.org/licenses/>.
 */

/**
 * \file   create.c
 * \brief  Implementation of algorithms to generate perfect mazes.
```

```
 * \author Laperche Sylvain, <sylvain.laperche@gmx.fr>
 * \author Mouelet Leandre,  <**********@*****.**>
 */
```

1.3 Gardiens

Les gardiens seront de la forme H_NOMDUFICHIER_AAAAMMJJ afin de garantir leur unicité.

2 Code

2.1 Langue

Le code (nom des fonctions, nom des variables, etc) et les commentaires devront être en anglais.

2.2 Indentation

La tabulation ne doit pas être utilisée. L'indentation sera donc réalisée avec des espaces (4 espaces par niveau).

2.3 Saut de ligne

Un saut de ligne séparera la déclaration des variables en début de fonction de la suite du code. De même, chaque bloc (if, if-else, while, for, etc) sera suivi d'un saut de ligne si des instructions sont placées à sa suite.

En dehors de ces cas, les sauts de lignes seront utilisés à chaque fois qu'ils améliorent la lisibilité.

2.4 Alignement horizontal

L'alignement horizontal sera utilisé lorsqu'il facilite la lecture du code.

2.5 Espacement horizontal

Les opérateurs unaires sont collés à leur opérande.

```
!value
~bits
++i
j--
(unsigned int)x
*ptr
&x
sizeof(x)
```

Les opérateurs binaires (et le ternaire) sont séparés de leurs opérandes par une espace.

```
c1 = c2;
x + y
i += 2;
n > 0? n : -n
a < b
c >= 2
```

Les mots clefs **if**, **while**, **for** et **switch** ne sont pas suivi par une espace.

```
if(a > b)
while(x > 0)
for(i = 0; i < 10; i++)
switch(x)
```

2.6 Accolades

Les accolades sont utilisées même lorsqu'elles sont facultatives afin d'éviter des erreurs bêtes (oublier de les ajouter en cas d'évolution du corps de la structure de contrôle).

```
if(var == 42)
{
    puts("yeahh");
}
```

2.7 for, while et do-while

```
for(i = 0 ; i < 42 ; i++)
{
    foo(i) ;
}

while(i < 42)
{
    foo(i) ;
}

do
{
    foo(i) ;
} while(i < 42) ;
```

2.8 if et else

```
if(neighbors && !hunt)
{
    foo() ;
}
else
{
    bar() ;
}
```

2.9 switch

```
switch(dir)
{
    case NORTH :
        do_something() ;
        break ;
    case EAST :
        do_something() ;
        break ;
    case SOUTH :
```

```
        do_something() ;
        break ;
    case WEST :
        do_something() ;
        break ;
    default :
        break ;
}
```

2.10 Opérateur virgule

On n'utilisera pas l'opérateur virgule pour enchaîner plusieurs instructions.

2.11 Ligne de code

En général, on n'effectuera qu'une seule action par ligne de code (comme déclarer une seule variable par ligne). Là encore, la visibilité est en jeu.

3 Variable

3.1 Déclaration

Le qualificateur '*' s'applique au type (on déclare un pointeur d'entier par exemple, c'est un type à part entière). On utilisera donc la syntaxe suivante.

```
int* foo ;  /* OK.      */
int *foo ;  /* Pas OK ! */
int * foo ; /* Pas OK ! */
```

3.2 Initialisation

Toutes les variables devront être initialisées à leurs déclaration avec une valeur qui à du sens et si la suite de l'algorithme en dépend, dans le cas contraire, leur initialisation est reportée au moment adéquat. Les variables de parcours de boucles **for**, elles, seront initialisées dans l'en-tête de la boucle.

```
int error = 0;
int end = 0;
int i;
int max = maze->m * maze->n;

for(i = 0; i < max; i++)
{
    do_something(i);
}
```

3.3 Variables globales

En général, les variables globales sont une très mauvaises idées, et il sera bon de limiter leur usage au strict minimum. S'il s'avère qu'une variable globale est nécessaire elle devra :

- être déclarée au début du fichier dans une zone spécialement dédiée à cet effet ;
- être explicitement initialisée ;
- Justifiée et très bien documentée par des commentaires.

3.4 Utilisation

Il ne devra subsister aucune variables ou constantes inutilisées dans le code.

4 Structures de contrôle

4.1 Affectation

Il n'est pas recommandé de faire des affectations dans les conditions. Ainsi, on préférera

```
file = open(filename, "r");

if(file)
{
.....
}
```

À

```
if(file = open(filename, "r"))
{
.....
}
```

On est pas payé à la ligne ;-). Dans le cas où une affectation serait justifiée, on l'entourera de parenthèses pour ne pas faire croire à un test d'égalité foiré (de toute façon le compilateur émettra un avertissement).

```
while((c = getchar()))
{
    ... DO SOMETHING WITH c ...
}
```

4.2 Boucle for

On ne fera pas de boucle **for** avec un corps vide, autant utiliser un **while**.

Seule les variables d'itération seront initialisées et incrémentées (ou décrémentées) dans l'en-tête du **for**. Les autres variables seront manipulées dans le corps.

4.3 goto et ses amis

On limitera au maximum l'utilisation de **continue**, **goto** et **break** (sauf dans les **switch** pour ce dernier) sauf si leur usage apporte un gain réel et quantifiable au code sans que cela nuise à sa lisibilité.

5 Commentaire

5.1 Style

Seul les commentaires C89 (donc /* et */) seront utilisés. Si possible, les commentaires seront alignés.

Si un commentaire commente une instruction, il sera placé à la suite de celle-ci (dans la limite des 80 caractères). S'il commente un bloc, il sera placé au-dessus.

```
new_size = old_size * GROW_RATE;    /* compute the new size */
alloc_size = new_size - old_size;  /* find the needed new memory */

/* If we have more than 3 foobar, frob them */
if(x > 3)
{
    ... frob  foobar ...
}
```

Pour les commentaires multilignes, on adoptera la présentation suivante.

```
/*
 * Bla bla
 * frob foobar
 * do something
 * 42
 */
```

5.2 Code commenté

On ne laissera pas du code commenté dans les fichiers sources, si on utilise un gestionnaire de versions c'est pour, entre autres, éviter ce genre d'horreurs (ça fait crade, gêne la lecture, ...).

6 Conventions de nommage

variable	à base d'underscore	toto, foo_bar
fonction	à base d'underscore	foo(), foo_bar(), delay_ms(), kbd_get_char()
constante	majuscule avec underscore si nécessaire	TOTO, FOO_BAR
structure	première lettre en majuscule	List, Stack
enumeration	première lettre en majuscule, valeurs en majuscule	typedef enum Bool { FALSE, TRUE } Bool ;
union	première lettre en majuscule	
typedef	première lettre en majuscule	

On évitera tous les mélanges douteux (Camel Case par endroit, à base d'underscore à d'autres).

6.1 Liste d'abréviations couramment utilisées

argument	arg
buffer	buf
clear	clr
clock	clk
compare	cmp
configuration	cfg
context	ctx
delay	dly
device	dev
display	disp
error	err
function	fct
hexadecimal	hex
initialize	init
mailbox	mbox
manager	mgr
maximum	max
message	msg
minimum	min
Operating System	os (OS)
overflow	ovf
pointer	ptr
previous	prev
priority	prio
read	rd
ready	rdy
schedule	sched
semaphore	sem
stack	stk
synchronize	sync
timer	tmr
trigger	trig
write	wr

6.2 Unité

Si les valeurs retournées par une fonction ont une unité, celle-ci devra être précisée dans le nom de la fonction (en tant que suffixe).

```
int delay_ms(unsigned int t);
```

7 Fonctions

7.1 Déclaration

Un prototype devra être fourni pour chaque fonctions et des identificateurs seront donnés à chaque paramètres. Les prototypes devront être conformes à la norme ANSI.

```
void foobar(void);              /* OK.      */
void barfoo();                  /* Pas OK! */
```

Là encore, on veillera à respecter les conventions sur le placement de * et sur l'espacement horizontal.

```
int* foobar(char* str, int sz);  /* OK.      */
int * barfoo(char *str,int sz);  /* Pas OK! */
int *barfoo(char * str, int sz); /* Pas OK! */
```

Les types et les identificateurs des paramètres devront être identique dans le prototype et dans la définition.

7.2 Définition

Les variables locales à la fonction sont toutes déclarées au début de la fonction, un saut de ligne séparera ce « bloc » du reste du code. De même pour les blocs (les variables locales à un bloc seront déclarées au début de celui-ci).

On ne jouera pas avec la portée des variables (masquer une variable avec une autre) et on limitera les **return** à 1 seul par fonction (facilite le debug et ne « casse » pas le flot d'instructions).

Les fonctions devront être courtes. 80-100 lignes semble être un bon maximum (quelques exceptions pourront être tolérés si elles sont justifiées).

7.3 En-tête

L'en-tête sera présenté de la manière suivante :

```
/**
 * \fn   int hunt_and_kill(mazeCS_s* maze, int freq, inout_e in, inout_e out)
 *
 * \brief Genere un labyrinthe en utilisant l'algorithme "Hunt & Kill".
 *
 * \param maze Le labyrinthe de type mazeCS.
 * \param freq Frequence d'entree en mode chasse (permet de diminuer le facteur
 *             riviere).
 * \param in   Type d'entree.
 * \param out  Type de sortie.
 *
 * \return Statut, 0 si tout est OK, sinon 1.
 */
```

On pourrait légèrement modifier la syntaxe afin d'obtenir des commentaires pour Doxygen.

7.4 Type de retour

Le type de retour devra être précisé explicitement (pas de **int** implicite). Si la fonction ne renvoie rien, elle devra être déclarée **void** et ne pas contenir de **return**.

7.5 Appel

Il n'y a pas d'espace entre le nom de la fonction et la parenthèse ouvrante.

```
foo();
```

Il y a au moins une espace après chaque virgule qui sépare les paramètres.

```
foo(a, b, c);
```

Si une fonction possède trop de paramètres, ces derniers seront séparés par des retour à la ligne et seront alignés.

```
spi_create(0,
           PC,
           (1 << 4) | 3,
           2,
           (1 << 4) | 1,
           0,
           SPI_CPHA0 | SPI_CPOL0,
           SPI_MSBFIRST,
           0);
```

8 Compilation

On veillera à compiler le code avec au moins les flags **-ansi**, **-pedantic**, **-Wall**, **-Wformat=2**, **-Winit-self** et **-Wwrite-strings**.

Si les bibliothèques externes le permettent (*i.e* ne génère pas trop de warnings) on pourra ajouter **-Wextra** et **-Wstrict-prototypes**. Si ce n'est pas possible, on les ajoutera juste dans le mode debug pour détecter les erreurs potentielles (mais pas dans le mode release pour ne pas polluer l'utilisateur final).

Pour le mode debug, on pourra ajouter **-g** et **-Wunreachable-code**.

Bien entendu, chaque warning provenant de notre code devra être considéré comme une erreur et devra donc être corrigé.