Guide des bonnes pratiques de programmation

Introduction

L'objectif de ce guide est de présenter / rappeler les bonnes pratiques à adopter en programmation afin de réaliser des programmes lisibles et donc exploitables par des tiers. Le formalisme retenu ici ne constitue qu'une des conventions de programmation existantes. Les différents exemples sont exprimés dans le langage de programmation C, mais ils s'appliquent généralement à tout langage. D'autre part, écrire un programme en respectant ces consignes ne garantit en rien la qualité de la structuration du code, qui elle dépend du patron de conception adopté.
En ne respectant pas les commandements présentés dans ce guide, vous vous exposez aux foudres des enseignants.

Commandements

1. **Commenter son code** : tu devras, avant même l'écriture d'une fonction ou d'un bloc d'instructions, détailler l'utilité de la fonction (ou du bloc d'instructions). Dans le cas d'une fonction, l'ajout supplémentaire d'un descriptif des paramètres ainsi que de la valeur de retour de la fonction paraît une bonne idée. Le formalisme à adopter est celui utilisé par l'outil **doxygen**[1] qui permet de générer automatiquement de la documentation à partir d'un code source. De même, vous devrez ajouter un en-tête descriptif dans chaque fichier source. Astuce : privilégiez les commentaires en anglais.

   Exemple d'en-tête :

```
/**
 * @file    file_name.c
 * @author  John Doe <jdoe@example.com>
 * @version 1.0
 *
 * @section LICENSE
 *
 * This program 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 2 of
 * the License, or (at your option) any later version.
 *
 * This program 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 at
 * http://www.gnu.org/copyleft/gpl.html
 *
 * @section DESCRIPTION
 *
 * This file contains all functions to manipulate stacks
 */
```

1. http ://doxygen.org

Exemple générique de la description d'une fonction :

```
/**
 * <A short one line description>
 *
 * <Longer description>
 * <May span multiple lines or paragraphs as needed>
 *
 * @param  Description of method's or function's input parameter
 * @param  ...
 * @return Description of the return value
 */
```

Exemple concret :

```
/**
 * Exponentiation
 *
 * Calculate the exponentiation of a number a^n
 * the exponent must be a positive integer
 *
 * @param number the base for the exponentiation
 * @param exponent the exponent for the exponentiation
 * @return an integer set to the result of the exponentiation
 */
int power(int number, int exponent)
{
    ...
}
```

2. **Indenter son code** : tu devras, à chaque nouveau bloc d'instruction, introduire un retrait par rapport à la ligne précédente. Les nouveaux blocs d'instructions débutent généralement à la suite d'une accolade ouvrante. La taille du retrait sera toujours de 4 espaces.

   Exemple : le code suivant n'est pas indenté

```
 if (unlikely(prev->policy == SCHED_RR))
 {
 if (!prev->counter)
 {
 prev->counter = NICE_TO_TICKS(prev->nice);
 move_last_runqueue(prev);
 }
 }
```

   Après indentation, il deviendra :

```
 if (unlikely(prev->policy == SCHED_RR))
 {
     if (!prev->counter)
     {
         prev->counter = NICE_TO_TICKS(prev->nice);
         move_last_runqueue(prev);
     }
 }
```

3. **Placer correctement les accolades** : tu devras, à chaque nouveau bloc d'instructions (fonction, boucle, etc.), délimiter ce bloc par des accolades. Elles devront être placées sur une ligne seule et sur la même colonne. Tu devras utiliser des accolades même pour une unique instruction.

   Exemple d'un mauvais placement des accolades :

```
for(i=0; i<n, i++) {
    ...
}

for(j=1; j<k; j++)
    printf("salut\n");

while(i>n)
    {
        ....
}
```

   Après correction :

```
for(i=0; i<n, i++)
{
    ...
}

for(j=1; j<k; j++)
{
    printf("salut\n");
}

while(i>n)
{
    ....
}
```

4. **Respecter une taille de ligne maximale** : tu devras t'assurer qu'aucune ligne de ton programme ne fasse pas plus de 80 caractères. Au delà de cette limite, tu poursuivras ton instruction sur la ligne suivante. Si nécessaire, tu appliqueras la coupure après un opérateur.

```
 printf("Nombre d'années : %d%d, Nombre de mois : %d\n", years1, ...
```

deviendra (par ex.) :

```
 printf("Nombre d'années : %d%d, "
        "Nombre de mois : %d\n",
         years1, years2, month);
```

5. **Déclarer correctement les variables** : tu devras déclarer tes variables au début de chaque fonction ou bloc d'instructions. Ne déclare qu'une variable par ligne. Initialise chaque variable directement par la valeur adéquate (NULL dans le cas d'un pointeur). Jamais, au grand jamais tu n'utiliseras de variables globales.
   Exemple de mauvaises déclarations de variables :

```
int main(void)
{
    int val, num1, wcount = 0;
    char *table, *printcolor;

    if(wcount == 1)
    {
        ...
    }

    float result;
    ...
```

   Les mêmes variables déclarées correctement :

```
int main(void)
{
    int val    = 0;
    int num1   = 0;
    int wcount = 0;

    float result = 0;

    char *table      = NULL;
    char *printcolor = NULL;

    if(wcount == 1)
    {
    }
    ...
```

6. **Découper son code en petites fonctions** : tu devras agencer ton code en petites fonctions simples et bien spécifiques. Plus une fonction est conséquente, moins elle devient compréhensible. Ta fonction principale (main) devra se trouver au début du fichier principal de ton code.

7. **Découper son code en plusieurs fichiers** : tu devras découper ton code en plusieurs fichiers de manière à ce que chacun d'eux regroupe les fonctions spécifiques à un thème. De plus, tu devras utiliser un fichier en-tête (généralement .h) qui comportera les différents *include*, définition des structures et prototypes des fonctions.

8. **Compiler régulièrement son code** : tu devras recompiler ton projet à chaque ajout de fonctions ou de blocs d'instructions. Après chaque compilation, tu devras lire et comprendre les messages affichés par le compilateur. Le cas échéant, tu devras directement corriger les erreurs (erreurs de syntaxe, warnings, etc.) relevés par le compilateur.

   Exemple d'une compilation avec erreurs :
   – mauvaise assignation d'une variable.
   – utilisation d'un objet non-défini.
   – utilisation d'une fonction pour laquelle le prototype est manquant.

```
gcc -Wall -o ploup ploup.c
ploup.c: In function 'main':
ploup.c:14: attention : assignment makes integer from pointer without a
            cast
ploup.c:16: erreur: 'SOCK_DGRAM' undeclared (first use in this function)
ploup.c:17: attention : implicit declaration of function 'socket'
```

9. **Tester régulièrement son code** : tu devras, après chaque modification de ton code, tester (e.g. lancer le programme) la nouvelle partie pour vérifier qu'elle fonctionne comme envisagé. Il faudra également vérifier que le nouveau code ne modifie pas en mal le comportement du reste du programme.

10. **Utiliser les outils à disposition** : tu devras utiliser les outils à ta disposition et adaptés à la situation.
    – Ecriture du code : utilisation d'un éditeur adapté tel que *Vim* [1], *Emacs* [2], *Anjuta* [3] et *Kate* [4], ou d'un IDE (Integrated Development Environment ou Environnement de Développement Intégré) tel que *Eclipse* [5] pour les projets plus conséquents.
    – Compilation du code : utiliser un moteur de production tel que *make*[6], *waf*[7] ou *SCons*[8].
    – Correction du code : utiliser un débuggeur tel que *gdb*[9] et un analyseur de l'utilisation de la mémoire tel que *valgrind*[10].
    – Sauvegarde du code : utilisation d'un gestionnaire de version tel que *subversion*[11], *GIT*[12] ou *Mercurial*[13].

Références

[1] The Vim editor, `http://www.vim.org`.
[2] The GNU Emacs editor, `http://www.gnu.org/software/emacs/`.
[3] The Anjuta editor, `http://projects.gnome.org/anjuta/index.shtml`.
[4] The Kate editor, `http://http://kate-editor.org`.
[5] The Eclipse IDE, `http://www.eclipse.org`.
[6] GNU Make `http://www.gnu.org/software/make/`.
[7] Waf building system, `http://code.google.com/p/waf/`.
[8] SCons building system, `http://www.scons.org`.
[9] The GNU Project Debugger, `http://www.gnu.org/software/gdb/`.
[10] Valgrind memory analyser, `http://valgrind.org`.
[11] The version control system subversion, `http://subversion.tigris.org`.
[12] The fast version control system, `http://git-scm.com/`.
[13] The distributed source control management tool Mercurial, `http://mercurial.selenic.com`.