# TD PG106
# Mise au propre d'un code source

Important : Pour ce TD, vous **devez** utiliser un éditeur de texte basique (vi, vim ou emacs). L'utilisation de Atom, vscode etc... est interdite.

## 1 Convention de code

Introduction :

Une "convention de codage" (coding style et naming convention) est un document qui décrit un certain nombre de régles quand à l'écriture de programme. Ce document fixe par exemple la langue (français/anglais), le nommage des structures (majuscule, minscule, utilisation de l'underscore), des macros, des fonctions, des variables locales et globales. Egalement, on y trouve des régles générales : type de retour de fonction, gestion des erreurs, position de l'accolade.

▸**Exercice 1.** `clang-format` *:*

*À quoi sert le programme* `clang-format` *? Quels sont les conventions supportées nativement par cet outil ? Parmis ces conventions, donner au moins 5 différences entre elles. (nous reviendrons par le suite sur cet outil).*

*Expliquer comment faire en sorte que la tabulation corresponde à 4 espaces et que les lignes ne fassent pas plus de 40 caractères.*

▸**Exercice 2.** *Mise en place d'un répertoire de projet :*

*Créer un répertoire nommé pg106, ce sera la racine de votre projet. Dans ce répertoire créer un fichier REAME.md.*

*Le fichier REAME.md sera rédigé au format Markdown (chercher sur internet). Dans ce fichier écrire une section coding convention dans laquelle vous décrirez l'ensemble des régles d'écriture de code pour votre projet. Vous pouvez bien entendu partir d'une norme existante et modifier celle-ci si besoin. Dans ce cas, pensez à donner l'url vers la convention en question.*

## 2 Remaniement de source

▸**Exercice 3.** *Récuperer l'archive hash.c.gz en ligne.*

*Décompresser le fichier à l'aide du programme gunzip. Ce source est une implémentation des tables de hachage. Le principe est de stocker des éléments dans la table de manière efficace de façon à ce que la recherche d'éléments se fasse rapidement.*

*Pour cela, une table de hachage est composée de N listes chainées. On ne connait pas la nature exacte des éléments de la table mais on dispose d'une fonction qui renvoie un entier associé à un élément. Cet entier détermine la liste dans laquelle l'élément doit être inséré. On dispose également d'une fonction qui permet la comparaison de deux éléments. Ainsi, les listes sont chainées dans l'ordre croissant des éléments.*

— *Quelle structure encode la liste chainée ?*

— *Quelle est le type de liste chainée ?*

— *À quoi correspondent les paramètres de hash_init ?*

*Dans votre répertoire de projet, créer un répertoire thirdparty et dans ce répertoire le répertoire hash qui contiendra le fichier source.*

Nous allons maintenant, étape par étape, mettre au propre ce code source.

▸**Exercice 4.** *Mise en forme de code syntaxique.*

*Utiliser* clang-format *pour la mise en forme syntaxique du programme.*

▸**Exercice 5.** *Homogénéisation du nommage des fonctions.*

*Commencer par renommer les fonctions en respectant votre convention de codage. Ajouter des commentaires devant chaque fonction avec une description en une ligne de ce que fait la fonction.*

▸**Exercice 6.** *Découpage.*

*Le source contient une implémentation de table de hachage et un programme d'exemple. Séparer les deux.*

▸**Exercice 7.** *Le header*
*Pour pouvoir garantir que le programme et l'implémentation des tables parlent de la même façon, il faut écrire un fichier d'entête.*

*Créer l'entête. Ajouter dedans les fonctions, structures etc... nécessaires.*

*Au niveau de la macro HASH_SIZE, celle-ci doit-elle aller dans le header ou non ? Faire en sorte que sa valeur puisse être redéfinie au moment de la compilation.*

# 3 Compilation

▸**Exercice 8.** *La compilation :*
*Compiler les deux fichiers sources en objets puis lier les deux en un programme exécutable. Vérifier que le programme fonctionne.*

▸**Exercice 9.** *Les asserts :*
*Ecrivez un deuxième programme d'exemple dans lequel vous appellerez la fonction d'ajout en spécifiant NULL comme premier argument. Compiler et exécuter : que se passe-t-il ?*
*Compiler à nouveau en ajoutant l'option* `-DNDEBUG`*. Executer : que ce passe-t-il ?*
*Consulter la documentation de la fonction assert.*

▸**Exercice 10.** *Bibliothèque :*
*Compiler le module hash sous la forme d'une bibliothèque static libhash.a et une bibliothèque dynamique libhash.so.*

*Compiler le programme d'exemple en utilisant l'une et l'autre de ces bibliothèques (cela produit donc 2 executables différents). À l'aide des programmes nm et objdump, lister les symboles présents dans vos exécutables. Utiliser le programme ldd pour vérifier les dépendances.*

*Modifier la fonction d'init pour qu'elle affiche une message sur la sortie standard. Recompiler les deux bibliothèque puis tester les executables précédents sans les recompiler.*

▸**Exercice 11.** *Script shell :*
*Ecrire un script shell qui automatise la complilation de vos sources. Votre script prendra en parametre "dynamic" ou "static" pour indiquer quel mode de compliation est souhaité.*

*Sachant que dans un script shell la ligne :*

```
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
```

*permet de stocker dans la variable DIR le nom de répertoire contenant votre script shell, ajouter la fonctionalité d'outsource building à votre script. Ceci signifie que tous les fichiers générés par la compilation (les .o, bibliothèques et executables) doivent être placé dans le répertoire courant et non avec les sources (sauf si le répertoire courant est le répertoire des sources bien sûre).*