TD PG106
cmake

1 synchronisation avec thor

▸**Exercice 1.** *depôt thor :*
Sur la page PG106 de thor, trouver l'url du dépôt qui vous est associé. Dans votre dépôt, ajouter le dépôt bare *de thor comme* remote *avec le nom* thor.

Effectuer un pull *: vous allez faire face à un soucis. Le premier point est de définir d'associer le* master *de votre dépôt à la branch* master *de* thor. *Pour cela utiliser la commande* branch –set-upstream=thor/master master *: on indique que les modifications faîtes sur* thor/master *doivent être associées à votre* master *local (il en est de même pour le* push*).*

Le deuxième point est de définir la stratégie par défaut en cas de conflit : utiliser git config pull.rebase false *pour définir la stratégie par défaut comme étant un* merge.

Le troisème point est que pour l'instant master *et* thor/master *n'ont pas d'histoire commune. Pour forcer* git *à faire le pull malgré cela, il faut faire* git pull –allow-unrelated-histories. *Cette option ne sera plus nécessaire par la suite car nous aurons créer un point d'histoire commune auquel* git *pourra se raccrocher.*

▸**Exercice 2.** *branches : En utilisant la commande* git branch -l -a*, trouver les branches existantes sur le dépôt* thor. *A l'aide de la commande* git checkout reference*, basculer dans cette branche.*

▸**Exercice 3.** *nouvelle branche... : On va réaliser la suite de ce TD sur* cmake *dans une branche que nous fusionnerons avec* master *une fois le travail effectué. Comme vous êtes plusieurs à travailler sur le dépôt* thor*, chacun des développeurs va créer une branche spécifique. A la fin du TD, l'un de vous fera un merge des travaux de ce TD dans master.*

2 CMake

Nous allons écrire les fichiers nécessaires pour automatiser la compilation de la bibliothèque de table de hachage.

Attention : *cmake* est prévu pour l'"out-source building" (on compile ailleurs que dans les sources), ne lancer pas la commande *cmake* depuis votre répertoire projet (sinon, il faudra faire le ménage de l'ensemble des fichiers et répertoires générés par cmake).

▸**Exercice 4.** *On reprend (si nécessaire).*
Reprendre le fichier `hash.c` *du premier td. A partir de ce fichier, constuire les fichiers* `hash.h`, `hash.c` *et* `exemple1.c`.

▸**Exercice 5.** *Ecriture du fichier cmake.*
`cmake` *repose sur l'écriture d'un fichier de description nommé* `CMakeLists.txt`. *Ci-dessous un exemple basic d'un tel fichier :*

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_library (hello hello.c)

target_include_directories (hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})

add_executable (demo demo.c)
target_link_libraries (demo LINK_PUBLIC hello)
```

A la racine de votre projet, écrire un fichier `CMakeLists.txt` *qui compile la bibliothèque ainsi que le programme d'exemple.*

Créer un répertoire `build`*, tester votre fichier CMakeLists.txt en utilisant la commande* `cmake ..` **depuis le répertoire** `build`*. Vous pouvez également effectuer cette étape avec l'outil graphique* `cmake-gui`*.*

▸**Exercice 6.** *Compilation :*
Une fois que l'étape de configuration de cmake s'est bien déroulée, vous pouvez compiler avec make *ou encore* make VERBOSE=1 *pour voir les commandes utilisées pour la compilation. Quel est le type de la bibliothèque créée ? En consultant la documentation de la fonction* `add_library` *de* cmake*, expliquez comment choisir le type de la bibliothèque (dynamique ou statique).*

▸**Exercice 7.** *Organisation des sources :*
Nous allons organiser nos sources de la façon suivante : dans un répertoire `hash` *nous mettrons les fichiers de la bibliothèque, dans un répertoire* `demo` *les programmes d'exemples. La commande :*

```
add_subdirectory(path)
```

permet d'indiquer à cmake *qu'il faut également charger le fichier* `CMakeLists.txt` *présent dans le répertoire* `path`*.*

Créer les répertoires `hash` *et* `demo`*. Déplacer les sources* `hash.c` *et* `hash.h` *dans* `hash` *et* `exemple1.c` *dans* `demo`*. Ajouter dans chacun des répertoires un fichier* `CMakeLists.txt` *qui contient les instructions adéquates.*

Vérifier que tout fonctionne correctement.

▸**Exercice 8.** *merge :*
Après avoir validé le bon fonctionnement de cet environnement de compilation, fusionner votre branche avec master *au niveau du dépôt.*

3 IDE

Un I.D.E est un environnement dédié à la programmation, nous allons utiliser QtCreator

▸**Exercice 9.** *QtCreator : Supprimer le répertoire de* build *créer précédemment : il est possible de faire en sorte que QtCreator utilise votre répertoire mais il est plus simple de le laisser gérer cela dans un premier temps.*

Lancer le programme qtcreator. Charger votre projet est utilisant l'action "ouvrir un fichier ou projet..." et en sélectionnant votre fichier `CMakeLists.txt` *(celui à la racine). Suiver les instructions à l'écran en choisissant une compilation "Desktop" : en ouvrant les détails, choisissez uniquement les modes Debug et Release.*

En utilisant la flèche verte en bas à gauche, tester la compilation et l'exécution de votre programme. L'affichage du résultat de la compilation et de la sortie du programme se font dans les consoles présentes en bas. Vous pouvez re-ouvrir des consoles en cliquant sur leurs noms en bas de la fenêtre. Il est possible que le programme se soit executé dans un terminal externe : dans ce cas, cliquez sur "Projects" dans la colonne de gauche puis sur "Run" (deuxième colonne à gauche) et décocher "run in terminal".

▸**Exercice 10.** *refactoring : Le* refactoring *consiste en la ré-écriture d'une partie d'un code source. Dans qtcreator, ouvrez le fichier* `hash.h` *(dans la liste en haut à gauche). Dans l'éditeur de code source, placer votre curseur sur le nom de la fonction* `HashAdd`*. Puis utiliser clique droit : refactor, renommer (ou bien le raccourci CTRL+SHIFT+R). Cela ouvre une boite en bas permettant de changer le nom de la fonction et de voir où ce nom est utilisé. Changer le nom en* `hash_add` *et vérifier que cette modification a bien été répercutée dans les fichiers* `hash.c` *et* `exemple1.c`*.*

▸**Exercice 11.** *Débug :*
Tout IDE propose un debuggage intégré. Cliquer sur la flèche verte-cafard pour lancer une session de debug.

En bas à gauche vous pouvez voir une icone représentant un ordinateur : c'est la cible de compilation actuelle. En cliquant sur cette icone, vous pouvez indiquer le mode de compilation et le nom de l'exécutable cible. Selectionner bien le mode débug et lancer une session de debuggage en utilisant la flèche verte-cafard (en bas à gauche).

Vous observez que le programme s'est terminé : nous n'avons pas introduit de point d'arrêt. Ouvrez le fichier `exemple1.c` *et au niveau de la déclaration du* `main` *cliquer dans la colonne à gauche du numéro de ligne. Un point rouge apparaît signalant un point d'arrêt.*

Relancer la session de debuggage. Vous pouvez maintenant debugger votre programme en utilisant les icones (ligne sous l'éditeur) ou les raccourcis clavier. Au milieu en bas vous avez la pile d'appel, en double cliquant sur le nom d'une fonction de la pile, vous pouvez aller dans son contexte. En bas à droite, vous avez les points d'arrêts définis. A droite, vous avez les variables de contexte.

Utiliser le clique droit sur les points d'arrêts ou bien les variables de contexte pour explorer les possiblités de debuggage. Pendant qu'une session de debug est active dans qtcreator, lancer la commande suivante dans un terminal : `ps -eaf`*. Sur quel outil repose de debuggage de qtcreator ?*

4 Doxygen

Nous allons maintenant ajouter de la documentation pour les tables de hachage en utilisant doxygen.
La documentation de doxygen est disponible à l'adresse : http://www.doxygen.org

▸**Exercice 12.** *Git :*
QtCreator intègre un support de plusieurs gestionnaires de sources dont git*. Dans le menu* Tools*, il y a un sous menu* git *permettant d'effectuer la plupart des commandes* git *: status, log etc...*

Via qtcreator ou bien dans un terminal, créer une nouvelle branche doxygen *dans laquelle nous allons faire la suite du TD.*

▸**Exercice 13.** *Mise en place.*
Créer un répertoire `doc`*. Dans ce répertoire, ajouter un fichier* Doxyfile *généré à l'aide de la commande* `doxygen -g`*.*

Éditer ce fichier afin d'indiquer que les sources à analyser sont situées dans le répertoire parent.

Depuis le répertoire doc, lancer la commande doxygen. Quels répertoires ont été créés ? Que contient la documentation ?

Modifier le fichier Doxyfile afin que seule la documentation html soit générée.

▸**Exercice 14.** *Ajouter la documentation des fonctions.*
En utilisant la documentation en ligne de doxygen, ajouter des blocs de commentaire pour chacune des fonctions du fichier `hash.h`*. Dans qtcreator, commencer un commentaire par /∗! devant le nom d'une fonction puis tapper la touche "entrer" : un bloc préremplit devrait appaître. Relancer doxygen régulièrement et vérifier que votre documentation apparait.*

▸**Exercice 15.** *Options :*
Trouver les options de doxygen qui permettent :
- *D'afficher le code source d'une fonction dans la doc de celle-ci.*
- *D'inclure les fonctions même non documentée.*

▸**Exercice 16.** *Doc de module :*
Trouver un moyen d'écrire une documentation de module dans `hash.h` *qui soit inclue dans la page de présentation de la documentation générée.*

▸**Exercice 17.** *Readme.*
Si vous n'en avez pas, écrire un fichier `README.md` *à la racine du projet. Vous utiliserez le format Markdown pour mettre en forme ce fichier. Il contiendra entre autre, le titre du projet, les auteurs du code et la licence.*

Faire en sorte que ce fichier génére le page de présentation de la documentation générée.