Travaux Dirigés Programmation C avancée

Informatique 1ère année.

—*Julien Allali* - allali@enseirb-matmeca.fr —

1 CMake

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

▸**Exercice 1.** *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 2.** *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 2.8.11)
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.*

Attention : *cmake est prévu pour l'"out-source building", 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).*

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

▸**Exercice 3.** *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.

2 IDE

Un I.D.E est un environnement dédié à la programmation, nous allons en tester 2 : QtCreator et CodeBlocks

▸**Exercice 4.** *QtCreator : 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 et exécuter cmake sans argument.*

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.

▸**Exercice 5.** *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 6.** *Débug :*
Tout IDE propose un debuggage intégré. Cliquer sur la flèche verte-cafard pour lancer une session de debug. Malheureusement, cela signal que l'exécutable n'a pas de symbole de débuggage : il n'a pas été compilé avec l'option `-g`*.*

Nous devons ajouter un nouveau contexte de compilation : celui pour débugger. Pour cela, cliquer sur l'icon "Projets" à gauche, s'affiche alors les différentes compilations du projet. Cliquer sur "Ajouter" (en haut au milieu) puis compilation. Entrer le nom debug. Indiquer l'option `-DCMAKE_BUILD_TYPE=Debug` *pour* `cmake`*.*

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.

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 ?*

3 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 7.** *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 8.** *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 9.** *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 10.** *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 11.** *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.

4 codeblocks

CodeBlock est un autre IDE disponible dans l'environnement de l'ENSEIRB-MATMECA. Nous allons voir comment utiliser cet IDE.

▸**Exercice 12.** *Générateur :*
`cmake` *repose sur le concept de générateur. Il peut construire un* `Makefile` *qui compilera vos sources en fonction des indications que vous avez fournies dans le fichier* `CMakeLists.txt` *: c'est ce qu'il fera pas défaut sous linux. Il peut également construire d'autres types de fichiers pour d'autre plateforme de compilation. Tapper la commande* `cmake -h`*. A la fin de l'aide, vous avez la liste des génrateurs supportés par cmake.*

Créer un repertoire codeblock. Depuis ce répertoire, invoquer `cmake` *avec la commande :* `cmake -G "CodeBlocks␣−␣Unix␣Makefiles"`*... Un fichier avec l'extension* `.cbp` *a été généré : c'est un projet codeblocks. Ouvrer ce fichier avec codeblocks.*

Une fois le projet chargé, cliquer sur "all" (en haut au milieu) et choisir "exemple1". Cliquer sur la flêche verte.