TD PG106
Git et Automatisation de la compilation avec Makefile.

git : config, commandes de base et les branches

Introduction :

Nous allons voir les commandes de base de `git` puis nous allons utiliser le système de branches. Les branches sont un point très important dans l'utilisation d'un dépôt (que ce soit avec `git` ou bien `svn` ou autre).

▸**Exercice 1.** `config` *:*

La commande `git config` *permet de positionner un certain nombre de propriétés et comportement de* `git`*. Ces propriétés peuvent être au niveau système, utilisateur ou spécifiques au dépôt. Chaque niveau masque le précédent : si une propriété est présente au niveau dépôt, elle prend le dessus sur le niveau utilisateur et système. Les propriétés dépôts sont dans le fichier* `.git/config` *du dépôt, celles au niveau utilisateur dans le fichier* `$HOME/.gitconfig` *et au niveau système dans* `/etc/gitconfig`*.*

Utiliser la commande `git config` *pour consulter lister les valeurs de configuration : à quel niveau sont positionnées ces valeurs ?*

▸**Exercice 2.** `status`, `add` *et* `commit`

Créer un fichier `LICENCE` *vide dans le dépôt (*`$> touch LICENCE`*) et vérifier que celui-ci est détecté par la commande* `git status`*. Ajouter le fichier au dépôt avec* `add`*, revérifier avec* `status` *et enregistrer cette modification au dépôt avec* `commit`*.*

Modifier le fichier `LICENCE` *et répéter l'opération d'enregistrement.*

Tester la commande `git log` *pour visualiser les modifiations passées : quel est l'auteur des modifications ?*

Comment modifier le message du dernier `log` *? Faîtes cette modification et observer l'impact sur l'indentifiant du* commit *(les 40 caractères après le mot* commit*).*

▸**Exercice 3.** *config suite :*

Faites une modificiation (commit) et vérifier dans `git log` *que le changement de configuration ci-dessus est bien actif.*

Quelle propriété permet de choisir l'éditeur de texte pour saisir des messages (lors d'un `commit` *par exemple) ?*

▸**Exercice 4.** *le répertoire* `.git` *:*

En listant le contenu du répertoire `.git`*, donner la fonction de chaque élément trouvé.*

Modifier la description du dépôt.

Faites en sorte que les fichiers `~` *et* `.o` *soient ignorés par défaut lors d'une appel à* `add` *ou* `status`*.*

▸**Exercice 5.** `gitk` *ou* `gitg`

`gitk` *(ou équivalent comme* `gitg`*) est un outil permettant de visualiser graphiquement les évolutions d'un dépôt git. Utiliser ce programme sur votre dépôt. Par la suite, utiliser ce programme pour avoir un visuel sur les modifications que vous faites sur les dépôts (penser à utiliser F5 pour mettre à jour gitk).*

▸**Exercice 6.** branch

Dans le dépôt créer une branche appelée `makefile` *avec la commande* `git branch -c makefile` *et vérifier avec* `git branch -l`*. Pour basculer dans cette branche, utiliser la commande* `git switch makefile` *(il est possible de combiner la création et le basculement avec* `git switch -c makefile`*).*

1 Makefile

Dans les exercices qui suivent, nous allons progressivement transformer notre script de compilation en un Makefile.

▸**Exercice 7.** *Makefile simple*

Commencer par reprendre chaque commande que vous avez mises dans votre script. Pour chaque commande, déterminer le fichier produit par la commande et ajouter dans le fichier Makefile une règle correspondante. Par exemple :

```
gcc -c list.c -o list.o
```

devient dans le Makefile :

```
list.o:
    gcc -c list.c -o list.o
```

Procéder ainsi pour l'ensemble de commandes de compilation présentes dans le script.

Une fois que l'ensemble fonctionne, enregistrer vos modifications dans la branche (commit).

▸**Exercice 8.** *dépendances : La question à se poser est : dans quels cas je dois recompiler une cible ?*

Si vous utilisez la commande `make list.o` *et que le fichier existe déjà, alors rien n'est fait. Si maintenant vous utilisez la commande* `touch list.c` *(cela va mettre à jour la date de modification du fichier) puis* `make list.o`*, rien n'est fait alors que l'on souhaite que le fichier soit re-compilé !*

Pour chaque règle, ajouter la liste des dépendances associées au fichier produit, par exemple :

```
list.o: list.c list.h debug.h
    gcc -c list.c -o list.o
```

Vérifier que cela fonctionne en utilisant successivement `touch` *et* `make`*.*

Procéder ainsi pour toutes les règles présentes dans votre `Makefile`*. Qu'est-ce qui permet de connaître les dépendances d'un fichier source ? Par exemple : pourquoi avoir ajouté* `debug.h` *dans les dépendances de* `list.o` *?*

▸**Exercice 9.** *coexistance entre bibliothèques statiques et dynamiques*

Il y a un petit soucis avec le fait de produire simultanément `libhash.a` *et* `libhash.so`*. En effet, les deux dépendent par exemple de* `hash.o` *mais dans le deuxième cas, il faut que celui-ci ai été produit avec l'option* `-fPIC`*.*

Ajouter des règles pour produire des `.o` *avec l'option* `-fPIC` *(on nommera ces fichiers* `nom_fpic.o`*) et mettez à jour les autres règles pour utiliser ces fichiers quand cela est nécessaire.*

Faites un commit pour enregister vos modifications.

▸**Exercice 10.** *all :*

Pour simplifier la compilation, nous allons créer une règle `all` *comme première règle du* `Makefile`*. En effet, lorsque l'on tappe* `make`*, alors c'est la première règle qui est choisi comme cible à produire.*

On notera que la cible `all` *ne produit pas un fichier* `all`*, on renseigne à* `make` *de la façon suivante au début du* `Makefile` *:*

```
.PHONY=all
```

Faites un commit pour enregister vos modifications.

▸**Exercice 11.** *clean : Ajouter une cible* `clean` *qui supprime tous les fichiers produits (.o, .so, .a . . .). Pensez à ajouter cette règle dans* `.PHONY`

Faites un commit pour enregister vos modifications.

pause git !

▸**Exercice 12.** *modification dans* `master`

Basculer dans la branch `master`. *Editer le fichier* `README.md` *et compléter la section* `About` *en ajoutant une ligne sur la composition du projet (table de hachage/programme de demo).*

Enregister cette modification (commit).

▸**Exercice 13.** *switch and merge*

Rebasculer dans votre branche `makefile` *et mettez cette branche à jour par rapport à* `master` *en utilisant la commande* `git merge master`

Visualiser votre dépôt avec `gitk`.

▸**Exercice 14.** *thor...*

Connecter vous à thor et aller sur la page de votre dépôt puis cliquer sur le visualisateur web (Web viewer).

Est-ce que vos travaux réalisés jusqu'à présent sont visibles ? Faire en sorte que ce soit le cas !

on repart sur Makefile !

▸**Exercice 15.** *variables : Si l'on souhaite compiler les souces avec une nouvelle option (*`-Wall` *ou* `-g`*), il faut éditer toutes les commandes de compilation !*

Pour simplifier cela, créer une variable `CFLAGS` *qui contient les options passées à* `gcc` *pour les phases de pré-compilation/compilation. Ajouter l'utilisation de cette variable dans les commandes concernées.*

Sur le même principe, créer une variable `LDFLAGS` *qui contient les option passées à* `gcc` *pour la phase d'édition de lien. Mettez à jour les commandes concernées.*

▸**Exercice 16.** *variables automatiques :*

Il existe des variables automatiques que l'on peut utiliser dans les commandes associées à une règle :

— `$@` *sera remplacée par la cible*

— `$<` *sera remplacée par la première dépendance*

— `$^` *sera remplacée par l'ensemble des dépendances.*

Mettez à jour vos commandes de production pour utiliser les variables ci-dessus. Au final, vos règles de production ne devrait plus contenir de nom de fichier explicitement.

Il existe d'autres variables automatiques que l'on trouvera ici

▸**Exercice 17.** *merge... :*

Enregistrer vos modifications dans votre branche (commit). On va re-intégrer les modifications dans la branche `master` *mais pour cela nous allons utiliser (depuis* `master`*)* `git merge --no-ff makefile`. *Quel est l'impact de cette option ?*

Terminer en visualisant les évolutions de votre dépôt dans `gitk`.