
# LOG1000 – Ingénierie logicielle

## TP #1 : GIT et Makefile

# Objectifs:

- Comprendre l'utilité et le fonctionnement des logiciels de gestion de versions.
- Apprendre à utiliser le logiciel de gestion de versions git avec ses diverses commandes et comprendre les différentes situations possibles lors du développement d'un projet avec plusieurs usagers.
- Comprendre l'utilité et le fonctionnement des fichiers de construction Makefile.
- Écrire un fichier Makefile de base pour construire et déployer un système en fonction des dépendances entre les éléments de construction.
- Compiler un projet open source en utilisant les commandes de Git appropriées.

# Déroulement du travail pratique (TP)

Les premières étapes de ce travail pratique vous permettront de paramétriser votre répertoire Git et de créer et copier les fichiers nécessaires à la résolution de la problématique. Vous devrez par la suite réaliser les exercices reliés à la problématique afin de valider que vous comprenez bien les principes derrière git. Une fois le projet logiciel de la problématique mis sur pied, vous devrez écrire un fichier de construction « Makefile » pour construire et déployer un logiciel à partir du code source. Vous devrez également montrer que vous êtes capable d'appliquer les concepts appris sur un projet open source de taille plus large.

Si vous n'avez pas encore fait la demande pour un répertoire git, veuillez former une équipe et communiquer à votre chargé de laboratoire, les noms,matricules et courriel polymtl associés aux coéquipiers. Les équipes sont constituées de 2 personnes. **Il est très important que les équipes soient formées avant le laboratoire afin d'être sûr que nous ayons le temps de créer votre répertoire.**

# Rédaction du rapport

Votre rapport devra être contenu dans un fichier nommé «rapport.pdf» qui doit être soumis à travers votre répertoire git (dans le dossier TP1, voir plus bas) . Tout y est indiqué, vous n'avez qu'à le copier et le coller afin de le remplir. N'oubliez pas de faire «git add», «git commit» et «git push» sur le rapport à la fin de votre TP pour vous assurer que le rapport soit soumis! Les détails sur comment faire ces commandes git vous seront divulgués au cours de la réalisation des différents exercices.

NB : Ne pas oublier de faire un "git ls-files" a la fin de votre tp et d'insérer la capture dans votre rapport.

# Partie 1:

# Mise en situation

Des études qui évaluent et classent les mots les plus courants en anglais examinent des textes écrits en anglais. L'analyse la plus complète est peut-être celle qui a été menée contre l'Oxford English Corpus (OEC), une très grande collection de textes du monde entier rédigés en anglais. Un corpus de textes est une vaste collection d'œuvres écrites organisées de manière à faciliter cette analyse.

Au total, les textes de l'Oxford English Corpus contiennent plus de 2 milliards de mots. L'OEC comprend une grande variété d'échantillons écrits, tels que des œuvres littéraires, des romans, des revues académiques, des journaux, des magazines, des débats parlementaires du hasard, des blogs, des journaux de discussion et des courriels.

On vous demande de faire un logiciel qui prend le nom d'un fichier comme argument, puis imprime à l'écran le mot le plus populaire dans ce fichier, ainsi que le nombre d'occurrences.

Dans notre cas, on va chercher le mot le plus populaire du livre « quantium_algo» (qui se trouve dans le dossier DATA joint a l'énoncé de ce tp). L'OEC pense que le mot "the" est plus populaire que "of". Notre rôle est de leur faciliter la tâche à l'aide de notre logiciel.

Le cœur du logiciel est composé de la classe "HashMap.cpp", qui représente une table de hachage (https://fr.wikipedia.org/wiki/Table_de_hachage), c.-à-d. une structure qui peut stocker des "mappings" d'une clef (ici un string) à une valeur (ici un entier). Par exemple, un HashMap avec les entrées ("abc",1) et ("def",2) retournera la valeur 2 si on passe la clef "def", tandis que rien ne sera retourné lorsque l'on donne la clef "ets" (car il n'y a pas d'entrée pour cette clef).

Le but de ce TP est d'améliorer une implémentation de base à partir du code source disponible sur Moodle. Avant de faire ces améliorations, il est nécessaire de mettre sur pied un répertoire de gestion des versions en utilisant git, puisque le code source existant était stocké sur une clé USB auparavant.

Pour le reste du TP, vous jouez le rôle d'Équipier 1 et Équipier 2. Il vous faudra spécifier au début de votre rapport (voir section: rédaction du rapport) qui entre vous sont réellement l'Équipier 1 et l'Équipier 2. Par exemple, Équipier 1: Paul Demers et Équipier 2: Joe Oliver. Au final, afin que le logiciel soit utilisable, il vous faudra développer un Makefile permettant de construire et déployer les outils.

# 1. Git

### Préparation du poste de travail et documentation

### Utilisation du «remote Shell»

Puisque vous avez seulement accès à 1 ordinateur par équipe, il vous faudra utiliser une petite astuce pour ouvrir le compte de plusieurs usagers sur une seule session Linux. Il suffit que le coéquipier n'ayant pas de session ouverte

ouvre une session à distance «remote Shell», soit de son laptop, soit sur le même poste informatique. Dans le dernier cas, il suffit d'ouvrir une nouvelle fenêtre du terminal et de saisir la commande suivante : «ssh nomusager@l4714-XX.info.polymtl.ca».

Remplacez «nomusager» par votre nom d'usager pour utiliser les ordinateurs de la polytechnique et «XX» par le numéro de votre poste informatique du laboratoire 4714. Le numéro du poste informatique est indiqué physiquement sur votre poste. Une personne avec un ordinateur personnel peut faire la même manœuvre. Si vous avez Windows, on recommande d'installer le logiciel libre VirtualBox avec une image virtuelle de Ubuntu Linux[1] pour obtenir un environnement similaire aux TPs.

### Documentation

La documentation git proposée dans le cadre du cours explique très bien comment réaliser les différentes étapes de ce laboratoire. La documentation est disponible au lien suivant: https://git-scm.com/book/fr/v2. Des informations rapides sont aussi disponibles par l'intermédiaire de la commande « git help » ou « git help *souscommande* ».

## E1.1 Mise en place de votre répertoire Git (3 pts)

Votre répertoire git sera bientôt ou est déjà en ligne si vous avez créé votre équipe en avance, et est disponible à un des URLs suivants: https://githost.gi.polymtl.ca/git/log1000-0X (si votre numéro de groupe < 10), https://githost.gi.polymtl.ca/git/log1000-XX (si votre numéro de groupe < 100) ou https://githost.gi.polymtl.ca/git/log1000-XXX (si votre numéro de groupe >=100). Remplacez les X par le numéro d'équipe que l'on vous a assigné sur https://docs.google.com/spreadsheets/d/1JLAfwVwIuC1FDVcnCdhYoaSnRUacYlJrWC0lox969wA/edit#gid=0. **L'un des coéquipiers** doit d'abord créer une copie locale du répertoire git et y ajouter le code source disponible sur Moodle dans le fichier d'archive **«TP1.zip»**.

Pour faire une copie locale du répertoire git, il vous faut exécuter la commande «git clone *URL_SERVEUR*» à partir du terminal pointant l'un de vos dossiers que l'on suggère de nommer LOG1000. Remplacez «*URL_SERVEUR*» par l'URL de votre répertoire git sur le serveur. Votre répertoire git sera également utilisé pour la remise de l'ensemble de vos travaux pratiques au courant de la session. Par la suite, copiez le contenu du fichier d'archive dans le répertoire local nouvellement créé et gardez la même hiérarchie de dossiers que celle du fichier d'archive.

Une fois que vous avez mis en place les dossiers et fichiers, vous devez exécuter, à partir du dossier contenant votre copie locale (dossier LOG1000 ou autre), la commande «git add TP1», suivie de «git commit -m "*votre message*"» et «git push» afin de propager les nouvelles modifications au serveur git. Prenez soin de mettre un commentaire pertinent et différent de «*votre message*» lorsque vous faites un «commit», la pertinence de vos commentaires sera évaluée (voir grille d'évaluation). L'équipier en question doit finalement exécuter la commande «git pull» pour s'assurer que les informations reliées au répertoire sont à jour.

À ce point, le deuxième coéquipier doit seulement faire un «git clone» en utilisant la même méthodologie que décrite précédemment (mais évidemment dans un autre dossier que l'Équipier 1). Le répertoire local git devrait contenir tous les fichiers que le premier coéquipier a «commit». Les équipiers 1 et 2 devraient avoir une copie locale du répertoire Git avec les mêmes fichiers sources.

**Assurez-vous que l'Équipier 1 et l'Équipier 2 aient les mêmes révisions des fichiers dans leur copie locale du répertoire Git!**

Finalement, l'un des 2 coéquipiers doit exécuter la commande «git log » et:

a) copier la sortie provenant du terminal dans le rapport sous la question E1.1. [/2]
Vous devriez ainsi avoir toute l'information concernant les nouveaux fichiers ajoutés au répertoire. Si ce n'est pas le cas, demandez de l'aide au chargé de laboratoire. Répondez également à la question

b) quelle est la différence entre les commandes «git log» et «git log -p» ? [/1]

# E1.2 Modification 1 (10 pts)

Les deux coéquipiers reçoivent la tâche de modifier trois fichiers du code source, soit les fichiers: «HashMap.h», «HashMap.cpp» et «main.cpp». Vous devez donc réaliser les étapes suivantes:

1. L'Équipier 1 dé-commente la signature de la méthode int compteur(const std::string& key) dans le fichier «HashMap.h»
2. Ensuite, il dé-commente  cette méthode en bas du fichier «hashMap.cpp».
3. Compilez manuellement le programme pour vérifier si le programme compile bien : « g++ -o pari *.cpp ».
4. Suite à la modification, l'équipier 1 doit exécuter la commande «git status». Répondez par la suite aux questions suivantes:
    a. Copiez et collez à votre rapport la sortie du terminal correspondante à l'exécution de la commande.[/3]
    b. Pourquoi le nom de l'exécutable, HashMap.h et HashMap.cpp sont écris en rouge dans la console? Est-ce que cette situation est normale? Pourquoi (pas)? [/1]
5. Pour terminer, l'équipier en question exécute les commandes: «git pull» ,«git add HashMap.h HashMap.cpp» suivi de «git commit -m "Votre Message"» et «git push». À ce point, une nouvelle révision de «HashMap.h» et «HashMap.cpp» devrait être créée.
6. L'Équipier 2 doit par la suite, sous son répertoire et sans faire un «git pull», modifier le fichier SomeKeyHash.cpp» dans la définition de la fonction «hash()». En particulier, il faut remplacer l'implémentation actuelle (qui toujours retourne 1) par l'implémentation djb2 de la page http://www.cse.yorku.ca/~oz/hash.html. Il faut quelques modifications pour que cela marche dans le contexte du logiciel pari. Compilez manuellement pour tester.
7. Afin de propager sa modification, l'Équipier 2 exécute les commandes: «git add» suivi de «git pull» «git commit -m "Votre Message"» et «git push» et une nouvelle révision de «SomeKeyHash.cpp» est ainsi créée. Répondez par la suite aux questions suivantes:
    a. Copiez la sortie du terminal correspondante à l'exécution de la commande «git pull».[/3]
    b. Est-ce que git a détecté un conflit? Pourquoi (pas)? [/2]
    c. Finalement, l'Équipier 2 doit exécuter la commande «git pull» (pour être certain que le «log» soit à jour) suivi de «git log --graph --decorate --all --oneline» et copier la sortie correspondante au rapport. S'il n'y pas de différence entre ce «log» et celui de E1.1, veuillez consulter le chargé de laboratoire. [/1]

# E1.3 Modification 2 (10 pts)

Vous êtes maintenant affectés à différentes tâches de modifications sur le fichier «main.cpp». Pour mieux avancer, les deux équipiers répartissent leur travail.

1. L'Équipier 1 et L'Équipier 2 prennent le soin de faire, en tant que bonne pratique, un «git pull» avant de débuter leurs travaux sur le fichier en question.
2. L'Équipier 1 doit modifier et sauvegarder le fichier «main.cpp» pour que la fonction main() prend comme argument le nom d'un fichier (un std::string) et peut imprimer chaque mot du fichier. Mettez cette implémentation avant la ligne « //utilisation normale », et gardez le reste de l'implémentation existante de la méthode main() comme tel. Regardez les articles suivants pour inspiration: http://www.cplusplus.com/articles/DEN36Up4/  et http://stackoverflow.com/questions/20372661/read-word-by-word-from-file-in-c.
3. Suite à la modification, l'équipier en question doit exécuter la commande «git status». Répondez par la suite aux questions suivantes:
    a. Copiez la sortie du terminal correspondante à l'exécution de la commande. [/2]
    b. Compilez le code source et copiez la sortie du programme. [/1]
4. En entretemps, l'Équipier 2 travaille aussi dans le fichier «main.cpp», mettant à jour les commentaires vides (/* */) au-dessus de la méthode main(). Ces commentaires doivent expliquer le but du logiciel pari.

5. Pour propager ses modifications, l'Équipier 1 exécute les mêmes actions que décrites précédemment (git add, git commit -m, git push) Après, pour propager les modifications, l'Équipier 2 exécute les mêmes actions que décrites précédemment Répondez par la suite aux questions suivantes:
    a. Copiez la sortie du terminal correspondante à l'exécution des commandes «git fetch ; git log --graph --decorate --all --oneline». [/2]
    b. Est-ce qu'il y aura un conflit lors d'un merge? Pourquoi (pas)? Si nécessaire, utilisez « git diff ..origin/master» pour décider. [/2]
    c. Maintenant faites « git merge », et copiez la sortie dans le rapport. [/2]
    d. Finalement, l'Équipier 2 doit exécuter la commande «git push» suivi de «git log » et copier la sortie correspondante au rapport . S'il n'y pas de différence entre ce «log» et celui de E1.2, veuillez consulter le chargé de laboratoire. [/1]

# E1.4 Modification 3 (10 pts)

On est presque là. Les deux équipiers font individuellement la révision cruciale pour finir le pari. Malheureusement, ils ne savent pas ce que l'autre équipier est en train de changer, ce qui pourrait être risquant.

1. L'Équipier 1 et L'Équipier 2 prennent le soin de faire, en tant que bonne pratique, un «git pull» avant de débuter leurs travaux sur le fichier en question.
2. L'Équipier 1 entame la première tâche de déclarer un HashMap « map » au début de main() (la ligne sous la signature de la fonction) et d'utiliser la méthode compteur() de la classe HashMap dans la boucle "for" ajouté pendant E 1.3 2) pour compter le nombre d'occurrences de chaque mot. Ensuite, enlevez l'ancien code de la méthode main() à partir de « //utilisation normale » jusqu'à la fin de la méthode. Compilez le code source résultant.
3. Pour propager ses modifications, l'Équipier1 exécute les mêmes actions que décrites précédemment en E1.2 3).
4. L'Équipier 2 décide en entretemps de déclarer un HashMap « mymap » au début de main() et de le parcourir dans une boucle for juste après la ligne « //utilisation normale » pour trouver et imprimer le mot avec le nombre d'occurrences le plus élevé. Utilisez la méthode getKeys() pour obtenir toutes les clefs du HashMap (utilisez vector<string> comme si c'était un tableau, la taille est disponible via la méthode size()), puis utilisez la boucle pour trouver la clef avec la valeur la plus élevée. L'Équipier 2 assume que l'Équipier 1 est en train d'écrire le code source qui remplira « mymap ». Notez que l'Équipier 2 n'enlève pas l'ancien code de main().
5. Pour propager ses modifications, l'Équipier 2 exécute les mêmes actions que décrites précédemment.(git add, git...)
    a. Copiez la sortie du terminal correspondante à l'exécution de la commande «git pull». [/3]
6. L'équipier 2 doit suivre les étapes suivante: [/5]
    - **git branch nouvelle-branche** (cette commande c'est pour créer une nouvelle branche git autre que MASTER sur laquelle on se trouve actuellement).
    - faire un **git checkout nouvelle-branche** (pour changer de branche, maintenant on va se trouver sur la branche nouvelle-branche, de ce fait toutes les modifications qui seront fait seront visible juste sur cette branche et pas sur la branche principale **master**).
    - se rendre sur le dossier du git( log1000-XX/TP1 ce chemin est un exemple et peut être différent d'une équipe a l'autre) est créer un fichier texte (textSurMaBranche.txt)
    - faire un **git (add textSurMaBranche.txt, commit -m "ajouter du fichier text sur la nouvelle branche")** (pour ajouter et propager le changement, l'ajout du fichier .txt" seulement sur la branche nouvelle-branche alors le changement n'est toujours pas visible sur la branche **master**)
    - **git checkout master** (pour revenir a branche **master** et la on va découvrir que les changement ne sont pas visible par les utilisateurs du master même si c'est le même utilisateur qui a fait le changement sur la nouvelle branche il sera dans l'incapacité de voir les changements sur cette nouvelle branche).
    - faire un **git merge nouvelle-branche**( on fait fusionner la branche créé avec la branche principale master pour propager les modifications précédemment faites sur la branche nouvelle-branche).

**Utilitée de cette exercise**: https://www.occitech.fr/blog/2014/12/un-modele-de-branches-git-efficace/

7. Décrire en quelques ligne que c'est il passé:
    a. Est-ce que Git a détecté un conflit? Pourquoi (pas)? Utilisez “git log --graph --decorate --all --oneline” et “git diff ..origin/master” pour supporter votre explication. [/3]
    b. Dans le cas où vous rencontrez une situation conflictuelle, comment pensez-vous régler cette dernière si on veut que le logiciel résultant réussit à résoudre le pari avec l’étudiant? Est-ce que ça peut être fait automatiquement? Pourquoi (pas)? Si oui, résolvez le conflit. [/3]
    c. Finalement, l’Équipier 2 doit exécuter la commande «git push» suivi de «git log -v» et copier la sortie correspondante au rapport. S’il n’y pas de différence entre ce «log» et celui de E1.3, veuillez consulter le chargé de laboratoire. [/1]
    d. [POINT BONUS] Quel est le mot le plus populaire dans « quantium algo » ? La bonne réponse doit montrer la sortie de votre programme, incluant la fréquence du mot gagnant. [/1]

# 2. Make

À ce stade du TP, vous avez acquis et validé vos connaissances sur la gestion de votre entrepôt de versions Git. Ayant eu des problèmes à manuellement compiler, générer et installer des programmes et des fichiers, vous voulez absolument un build system automatique et efficace. Pour faire le tout, vous devez utiliser un fichier Makefile.

## Documentation

D'abord, suivez attentivement ce tutoriel sur la base et les méthodes d’optimisation d'un Makefile:
https://www.youtube.com/watch?v=0YpgXKkNs04&list=PLyexDug1zljFuP8tC-PfhMIXtTKu1vFd3

## E2.1 Éléments de construction d’un exécutable (5 pts)

Avant même de rédiger votre Makefile, vous devez connaître l’ensemble des éléments nécessaires pour les deux phases de construction du logiciel: 1. compiler et 2. installer le système.

1. Pendant la compilation, l’exécutable que vous devez créer sera constitué de l’ensemble de code source, tenant compte des relations #include dans les fichiers code source. Le nom de l’exécutable est **pari.** Pour simplifier les choses, il vaut mieux sauvegarder tous les fichiers générés pendant la compilation dans un dossier séparé avec le nom **build/** (à côté des dossiers src/ et data/)**. On appellera cette phase « compile ».**

2. La deuxième phase du build (appelé « install ») crée un dossier **site/**, puis copie les fichiers générés par la compilation (de **build/**) ainsi que des fichiers .txt de **data/** (comme le livre « quantium_algo»[2]) vers le dossier **site/**.

Les deux équipiers veulent que le build soit automatisé complètement (sans activités manuelles à faire) et sera efficace, par exemple:

- Si on change le code source, les phases de compilation et installation devront être refaites pour que le système reste cohérent. Ce cas peut être testé avec la commande «touch src/HashMap.h; make» dans le terminal.

- Également, si on change un des fichiers .txt dans le dossier **data**, l’installation doit être refaite, mais sans recompilation du code source. Ce cas peut être testé avec la commande «touch data/quantium_algo.txt; make» dans le terminal.

Vous devez écrire de manière hiérarchique le graphe de dépendance des phases et des fichiers nécessaires pour exécuter le build comme décrit ci-dessus. Appelez la cible principale « all » (une convention populaire), c.-à-d. si on appelle « make » sans spécifier la cible, « all » sera choisie automatiquement. Écrivez votre réponse dans votre fichier

de réponses. À noter que même si actuellement vous n'avez pas les fichiers de dépendance .o, vous devez tout de même planifier leur intégration dans le graphe. Des cibles "phony", s'il y a lieu, doivent aussi figurer dans le graphe. [/5]

**Petit exemple de réponse:**

hello:
-start.o
	-start.cpp
	-hello.h
-hello.o
-hello.cpp
-hello.h

Ce que l'on peut voir dans cet exemple est que l'exécutable hello est fait du code compilé de start.o et hello.o. Le code source start.cpp appelle des fonctions de hello.o, la raison pour laquelle main.o dépend de hello.h. Si deux cibles dépendent d'une même cible, chacune doit mentionner cette dépendance dans votre graphe textuel.

## E2.2 Création du Makefile et exécution du programme (15 pts)

Vous devez maintenant rédiger un vrai **Makefile** en fonction des dépendances de construction que vous avez énumérées à l'étape précédente. Ne vous attardez pas à optimiser le script du **Makefile**. Suivez les étapes suivantes (astuce: ajoutez une commande "echo [un mot identifiant le règle]" dans chaque liste de commandes pour indiquer si les commandes de ce règle ont été exécutées):

1. Faire une mise-à-jour de votre copie locale («git pull») du répertoire Git d'équipe à partir du terminal de l'un des équipiers.
2. Créez et modifiez un fichier **Makefile** avec un éditeur de texte et sauvegardez-le sous le dossier TP1/ (**pas** dans src/). **N'oubliez pas les bonnes extensions de fichiers et d'utiliser le compilateur g++.** [/7]
3. Exécutez la commande ***make***.
4. Simulez des changements d'un fichier avec les deux commandes «touch ...» mentionnées ci-dessus. Copiez les deux sorties dans le rapport. [/3]
5. Faire «git add» du fichier **Makefile**, suivi d'un «git commit» et «git push» dans le répertoire Git.

**Astuce:**
Pour rendre le TP plus facile, ajoutez et complétez les deux cibles suivantes pour enlever les fichiers générés:

```
clean:
	#enlevez les fichiers générés

mrproper: clean
	#enlevez les dossiers générés
```

6. Expliquer le rôle et le fonctionnement des symbols, variables et commandes présents dans le makefile suivant: [/5]

```
CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
RM=/bin/rm
MAKE=/usr/bin/make
MAKEDEPEND=/usr/X11R6/bin/makedepend

SRC= a.c \
     b.c \
     c.c
OBJ=$(subst .c,.o,$(SRC))

SUBDIR= paf pof


.SUFFIXES: .c
.c.o:
        $(CC) -c $(CFLAGS) $<

all:
        for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
        $(MAKE) monProgramme

monProgramme: $(OBJ)
        $(CC) -o $@ $(LDFLAGS) $^

clean:
        $(RM) -f $(OBJ) core *~
        for i in $(SUBDIRS); do (cd $$i; $(MAKE) clean); done

depend:
        $(MAKEDEPEND) -- $(CFLAGS) -- $(SRC)
```

# Partie 2 : [/6]

Pour vous familiariser encore plus avec git , les commandes de compilation et pour se familiariser avec le code source du logiciel open source Ring[3] ,vous allez compiler une de ses parties en suivant les étapes ci-dessous :

1. Sortez du dossier que vous avez utilisé comme clône de votre répertoire git dans la partie 1, et créez un nouveau dossier (qui ne fera pas partie de votre propre répertoire) dans lequel vous allez exécuter cette commande : git clone https://gerrit-ring.savoirfairelinux.com/ring-daemon

Cela va vous permettre d'avoir une copie de la composante du logiciel ring sur laquelle vous allez travailler dans le TP2.

2. Pour cela, vous allez maintenant commencer les étapes de compilation :

*cd contrib*
*mkdir native*
*cd native*
*../bootstrap*
*make*

NB: Le make peut prendre quelques minutes pour terminer.

a. Maintenant exécutez la commande **«time make »** et insérez la capture de la sortie. [/2]
b. Changez le nom de l'attribut privé callIDSet_ partout dans les fichiers src/account.h et src/account.cpp. Puis, exécutez une deuxième fois la commande **«time make »** et insérez la capture de la sortie. [/2]
c. Est ce qu'il y a une différence de temps entre les deux exécutions de la commande? Pourquoi (pas)? [/2]

# Remise du travail pratique

## Considérations importantes pour la fin du TP

Toujours faire un «git add» des fichiers modifiés et un «git commit» suivie de «git push» de vos dernières modifications pour que l'on puisse voir la dernière version de votre travail lors de la correction. Si vous ne faites pas de commit, il se peut que l'on évalue une version différente (plus ancienne) de votre TP local sur le serveur Git.

### Vérification que vos travaux sont présents dans le répertoire Git du serveur:

Vous pouvez utiliser la commande «git *ls-files* *https://githost.gi.polymtl.ca/git/log1000-XX*» (ou une variation dépendamment de votre numéro de groupe) afin de voir les fichiers dans le dernier snapshot de l'entrepôt Git lui-même. Remplacez les X par le numéro de votre équipe. Ce que vous verrez dans cette liste correspond à ce que l'on verra pour la correction.

**!! DATE LIMITE DE REMISE !!**
**Voir sur moodle**

**Pénalités :**

**Non remise de rapport -50% de la note final après correction des fichiers remis séparément.**
**Pour retard - 10% par jour**
**Aucune remise par courriel n'est acceptée.**

Références:


1. http://www.psychocats.net/ubuntu/virtualbox
2. https://ring.cx/