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.
* Écrire un fichier Makefile utilisant les fonctions avancées et des astuces pour en simplifier le code.
* Compiler un projet open source en utilisant les commandes de Git

# Déroulement du travail pratique (TP)

Les premières étapes de ce travail pratique vous permettront de paramétrer 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’optimiser le script de votre « Makefile » afin d’assurer une bonne maintenance après.

Si vous n’avez pas encore fait la demande pour un répertoire git, veuillez former une équipe et communiquer à votre chargé(e) de lab les noms et matricules 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.txt» qui doit être soumis à travers votre répertoire git (dans le dossier TP1, voir plus bas). Voici le gabarit de référence que vous devez suivre pour rédiger ce rapport. 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! Les détails sur comment faire ces commandes git vous seront divulgués au cours de la réalisation des différents exercices :

Équipier1: *Prénom Nom*

Équipier2: *Prénom Nom*

*Partie 1 :*

E1.1

a)

b)

E1.2

2-a)

2-b)

5-a)

5-b)

Copie du log: *Insérer la copie de log demandée ici*

E1.3

3-a)

6-a)

6-b)

Copie du log: *Insérer la copie de log demandée ici*

E1.4

5-a)

5-b)

5-c)

Copie du log:*Insérer la copie de log demandée ici*

E2.1

a)

b) *Insérez votre graphe de dépendances ici*

E2.2

*Exécutez* «make»*, puis insérez la sortie des deux commandes* «touch ...; make»

*Partie 2 :*

# Pondération du travail pratique

Parie 1 : [ /46]

Pour E1.1 à E1.4, la qualité de vos messages pour les «commits»: [ /2]

Pour E1.1 à E1.4, la qualité du code source: [ /2]

E1.1 [ /3]

a) [ /2]

b) [ /1]

E1.2 [ /8]

2-a) [ /2]

2-b) [ /1]

5-a) [ /2]

5-b) [ /2]

Copie du log: [ /1]

E1.3 [ /8]

3-a) [ /3]

6-a) [ /3]

6-b) [ /2]

Copie du log: [ /1]

E1.4 [ /8]

5-a) [ /3]

5-b) [ /3]

Copie du log: [ /2]

E2.1 [ /5]

*Insérez votre graphe de dépendances ici* [ /5]

E2.2 [ /10]

Fichier Makefile [ /7]

*Exécutez* «make»*, puis insérez la sortie des deux commandes* «touch ...; make» [ /3]

Parie 2 : [ /4]

TOTAL: [ /50]

*Partie 1 :*

# Mise en situation

Tout au long de la session nous allons travailler sur un projet open source qui s’appelle Ring. Pour cela nous allons vous approcher de tout les outils nécessaire lors de développement de ce genre de projet logiciel. Premièrement, nous allons voir comment améliorer un outil pour les tests logiciels déjà existant pour générer les données des utilisateurs d’une application. Cet outil prend comme entrée (INPUT) des fichiers textuels contenant des listes de noms, de prénoms, de mails, et de numéros de téléphone pour chaque clients, et a pour but de générer des données personnelles réalistes à partir de ces fichiers d’entrée. Ensuite, un petit script Bash prend les données générées et leur transforme en une page HTML pour visualisation.

Évidemment, comme vous avez l’esprit ingénieur et que vous désirez fortement montrer vos compétences, vous allez faire de votre mieux pour réussir ce laboratoire. 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’Équipier1 et Équipier2. Il vous faudra spécifier au début de votre rapport (voir section: rédaction du rapport) qui entre vous sont réellement l’Équipier1 et l’Équipier2. Par exemple, Équipier1: Paul Demers et Équipier2: 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\_polytechnique@l4712-XX.info.polymtl.ca».

Remplacez «nomusager\_polytechnique» par votre nom d’usager pour utiliser les ordinateurs de la polytechnique et «XX» par le numéro de votre poste informatique du laboratoire 4712. 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]](#footnote-2) 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 (5 pts)

Votre répertoire git sera bientôt ou est déjà en ligne si vous avez créé votre équipe, et est disponible à l’URL suivante: [https://githost.gi.polymtl.ca/cours/log1000-XX](https://svnhost.gi.polymtl.ca/cours/log1000-XX) (Remplacez XX par le numéro d’équipe que l’on vous a assigné). **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 checkout *URL\_SERVEUR (nomDeLaBranche)*» 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, ajoutez 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 checkout» en utilisant la même méthodologie que décrite précédemment. 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’Équipier1 et l’Équipier2 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. 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 diff» et «git diff HEAD» ?

## E1.2 Modification 1 (10 pts)

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

1. L’Équipier1 doit modifier et sauvegarder le fichier «main.cpp» pour que les noms des fichiers «data/noms.txt», «data/prenoms.txt», «data/mail.txt», «data/téléphone.txt» et «data/data.txt» ne sont plus hardcodés, mais passés comme argument au programme.  
   Regardez l'article suivant pour inspiration:

http://www.cplusplus.com/articles/DEN36Up4/. Compilez manuellement le programme pour contrôler si le programme compile bien : « g++ -o executable \*.cpp ».

1. Suite à la modification, l’équipier doit exécuter la commande «git status -u». Répondez par la suite aux questions suivantes:
   1. Copiez et collez à votre rapport la sortie du terminal correspondante à l'exécution de la commande.
   2. Pourquoi y-a-t-il un point d'interrogation à côté du nom de l'exécutable que vous venez de compiler? Est-ce que cette situation est normale? Pourquoi (pas)?
2. Pour terminer, l’équipier en question exécute les commandes: «git pull» suivi de «git commit -m “Votre Message”» et «git push». À ce point, une nouvelle révision de «main.cpp» devrait être créée.
3. L’Équipier2 doit par la suite, sous son répertoire et sans faire un «git pull», modifier le fichier «utilitaires.cpp» dans la définition de la fonction «generezLignes()». En particulier, il faut ajouter un contrôle de l'argument «nr». Si cela a une valeur négative, la fonction doit quitter sans rien faire.
4. Afin de propager sa modification, l’Équipier2 exécute les commandes: «git pull» suivi de «git commit -m “Votre Message”» et «git push» et une nouvelle révision de «utilitaires.cpp» est ainsi créée. Répondez par la suite aux questions suivantes:
   1. Copiez la sortie du terminal correspondante à l’exécution de la commande «git pull».
   2. Est-ce que git a détecté un conflit? Pourquoi (pas)?
5. Finalement, l’Équipier2 doit exécuter la commande «git pull» (pour être certain que le «log» soit à jour) suivi de «git log » et copier la sortie correspondante au rapport sous la section «log». S’il n’y pas de différence entre ce «log» et celui de E1.1, veuillez consulter le chargé de laboratoire.

**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 répartiront leur travail.

1. L’Équipier1 et L’Équipier2 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’Équipier1 commence à implémenter la fonction «estDuplicat()» qui a pour but de retourner vrai si le pair d'arguments de la fonction a été rencontré déjà pendant l'exécution du programme. La fonction est dans le fichier utilitaires.cpp , il faut juste la compléter.
3. Suite à la modification, l’équipier en question doit exécuter la commande «git status -u». Répondez par la suite aux questions suivantes:
   1. Copiez la sortie du terminal correspondante à l'exécution de la commande.
4. En entretemps, l'Équipier2 travaille aussi dans le fichier «utilitaire.cpp», mais dans la définition de la fonction «generezLignes()». Là, il essaie d'appeler la fonction «estDuplicat()» (qui n'existe pas encore sur sa machine, mais qui est en train d'être développée par l'Équipier2) pour assurer que le pair <nom,prénom> généré arbitrairement par la fonction «ligneArbitraire» n'a pas encore été généré avant. Dans le cas où un «pair» est généré une deuxième fois, il faut refaire la génération pour obtenir un autre pair.
5. Pour propager ses modifications, l’Équipier1 exécute les mêmes actions que décrites précédemment en E1.2 3).
6. Pour propager les modifications, l’Équipier2 exécute les mêmes actions que décrites précédemment en E 1.2 3). Répondez par la suite aux questions suivantes:
   1. Copiez la sortie du terminal correspondante à l’exécution de la commande «git pull».
   2. Est-ce que git a détecté un conflit? Pourquoi (pas)?
7. Finalement, l’Équipier2 doit exécuter la commande «git update» suivi de «git log » et copier la sortie correspondante au rapport dans la section «log». S’il n’y pas de différence entre ce «log» et celui de E1.2, veuillez consulter le chargé de laboratoire.

**E1.4 Modification 3 (10 pts)**

On est presque là. Les deux équipiers font individuellement la révision du code source.

1. L’Équipier1 et L’Équipier2 prennent le soin de faire, en tant que bonne pratique, un «svn update» avant de débuter leurs travaux sur le fichier en question.
2. L’Équipier1 entame la première tâche et décide qu'il vaut mieux imprimer d'abord le prénom d'une personne, et après le nom. Pour échanger les deux champs, il faut changer le script «bin/data2html.sh» et les fonctions «generezLignes()» et «ecrivezLignes()» du fichier «utilitaires.cpp». Dans le script «bin/da- ta2html.sh», il y a une ligne qui mentionne "nom" et "prenom", où il faut juste changer l'ordre de ces deux mots. Ensuite, dans les deux fonctions mentionnées de «utilitaires.cpp», il y a une ligne où les termes «nom» et «prenom» doivent être échangés aussi. Compiler le code source.
3. Pour propager ses modifications, l’Équipier1 exécute les mêmes actions que décrites précédemment en E1.2 3).
4. L’Équipier2 décide peu de temps après d'enlever le mail de la sortie du programme. **Avant de commencer, il ne prend pas le soin de faire un «git add»** puisque cela a déjà été fait très récemment. De nouveau, il faut changer les deux fonctions de «utilitaires.cpp» mentionnées ci-dessus.
5. Pour propager ses modifications, l’Équipier2 exécute les mêmes actions que décrites précédemment en E 1.2 3). Répondez aux questions suivantes:
   1. Copiez la sortie du terminal correspondante à l’exécution de la commande «git add».
   2. Est-ce que Git a détecté un conflit? Pourquoi (pas)?
6. Finalement, l’Équipier2 doit exécuter la commande «git add» suivi de «git log -v» et copier la sortie correspondante au rapport sous la section «log». S’il n’y pas de différence entre ce «log» et celui de E1.3, veuillez consulter le chargé de laboratoire.

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. Veuillez suivre le gabarit imposé pour les réponses aux questions.

## 

## Documentation

D'abord, suivez attentivement ce tutoriel sur la base et les méthodes d’optimisation d'un Makefile: <http://gl.developpez.com/tutoriel/outil/makefile/>

Additionnellement, voici un peu plus d'explication sur des cibles "phony". Ce sont des cibles qui ne représentent pas des fichiers physiques, mais plutôt des activités de construction. Les commandes d'un règle avec une cible "phony" sont *toujours* exécutées, comme une fonction en C++, car une telle cible est **toujours** plus vieille que ses dépendances! Cette caractéristique les rend idéaux pour implémenter des build systems avec plusieurs phases, par exemple:

.PHONY: reussir\_bac annee1 annee2 annee3 annee4

reussir\_bac: annee1 annee2 annee3 annee4

echo "Wouhou!"

annee1: log1000.txt inf1010.txt

echo "Fini la premiere annee!"

annee2: log2000.txt

echo "Fini la deuxieme annee!"

#etc.

log1000.txt: intra.txt examen.txt

cat intra.txt examen.txt > log1000.txt

echo "Fini LOG1000"

#etc.

Si on exécute «make», la sortie sera toujours (comme les commandes des cibles "phony" sont toujours exécutées):

Fini la premiere annee!

Fini la deuxieme annee!

...

Wouhou

## Par contre, le message «Fini LOG1000» n'apparaîtra que si la cible «log1000.txt» doit être (ré)générée.

## 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 **executable.** 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/, bin/ et data/)**. On appellera cette phase « compile ».**

2. Après la compilation, il y a une phase de génération, où l'exécutable **executable** est utilisé pour générer un fichier **data.txt** à partir de 4 autres fichiers qui sont mentionnés dans le **Makefile** existant.

3. Finalement, la troisième phase du build utilise le script bin/data2html.sh pour générer la page **index.html** à partir de **data.txt**. Les commandes exactes pour faire ça sont mentionnées dans le **Makefile** existant. La page html est mise dans un dossier **site/** (dossier d'installation), et on copie aussi le dossier **data/css/** (avec le fichier **css/main.css**) vers ce dossier. Si vous ouvrez maintenant **index.html** dans votre navigateur web, il faudrait voir une page en couleur avec des données de personnes arbitraires.

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

● Si on change le code source de **executable**, les phases de génération de **data.txt** et l'installation (et géneration) de **index.html** devront être refaites (après la compilation) pour que le système reste cohérent. Recopier le dossier **css** n'est pas nécessaire. Ce cas peut être testé avec la commande «touch src/main.cpp; make» dans le terminal.

● Également, si on change un des fichiers.txt dans le dossier **data**, ces deux phases devront être refaites. Ce cas peut être testé avec la commande «touch data/noms.txt; make» dans le terminal.

● Si le fichier **data/index\_template.html** change, l'installation doit être refaite (génération de **index.html** et installation dans le dossier d'installation). Recopier le dossier **CSS** n'est pas nécessaire, sauf si le fichier **data/css/main.css** a changé. Ces deux cas peuvent être testés avec les commandes «touch data/index\_template.html; make» et «touch data/css/main.css; make» dans le terminal.

Questions:

a) Pourquoi le **Makefile** donné n'est pas bon? Donnez deux raisons.  
b) 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.

**Petit exemple de réponse:**

hello:

-start.o

-start.cpp

-hello.h

-hello.o

-hello.cpp

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 le graphe.

**E2.2 Création du Makefile et exécution du programme (10 pts)**

Vous devez maintenant rédiger un vrai **Makefile** en fonction des dépendances de construction que vous avez énumérées à l’étape précédente. Pour de la documentation, voir le lien proposé. Ne vous attardez pas à optimiser le script du **Makefile** pour cette étape. 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++.**
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.
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**

## *Partie 2 :*

Pour vous familiariser encore plus avec Git , les commandes de compilation et pour avoir une idée général sur une partie de code source Ring vous allez compiler une de ces partie en suivant les étapes si-dessous :

1) Dans le dossier que vous avez crée pour ce laboratoire vous allez exécuter cette commande : git clone https://github.com/savoirfairelinux/ring-daemon.git

qui va vous permettre d avoir une copie d une partie de code source  sur quoi vous allez travaillez une partie de TP2

2) vous allez maintenant commencer les étapes de compilation :

a) première partie

cd contrib

mkdir native

cd native

../bootstrap

make

b) deuxième partie

cd ../../

./autogen.sh

./configure

make

si vous voulez suivre les étapes a partir de site web voila le lien correspondent :

https://github.com/savoirfairelinux/ring-daemon

# Remise du travail pratique

## Considérations importantes pour la fin du TP

Toujours faire un «git add» des nouveaux fichiers 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 SVN du serveur:**

Vous pouvez utiliser la commande «git *list* [*https://githost.gi.polymtl.ca/cours/log1000-XX*](https://svnhost.gi.polymtl.ca/cours/log1000-XX)» afin de voir les fichiers dans l’entrepôt Git lui-même. Remplacez XX 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 !!**

**27 septembre 2016 à midi (12h)**

**Pénalités pour retard**

**10% par jour**

1. http://www.psychocats.net/ubuntu/virtualbox [↑](#footnote-ref-2)