

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 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'appliquer les concepts appris dans 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(e)** de laboratoire les noms et les matricules associés aux coéquipiers.

Les équipes sont constituées de **2 personnes**.

Rédaction du rapport

Votre rapport devra être contenu dans un fichier nommé « rapport.doc » qui doit être soumis à travers votre répertoire git (dans le dossier TP1, voir plus bas). Veuillez mettre vos réponses dans le fichier « Gabarit.doc » qui est fournis avec les fichiers sources. **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.

Partie 1 :

Mise en situation

Après avoir fait les cours de Log1000 et INF1010. Des étudiants ont décidés de créer un logiciel pour la détection de plagiat. Ce logiciel est basé sur plusieurs algorithmes, parmi ces derniers, un qui cherche le nombre d'occurrences d'un mot dans un texte.

Le prof de log1000 vous parle de ce projet, alors vous décider d'aider l'équipe de projet et vous faite un pari avec une autre équipe en ce qui concerne le mot le plus populaire dans la partie Hachage du livre «[The Art of Computer Programming](#), ». Votre prédiction est que le mot « dans » gagnerait, tandis que l'autre équipe prédit le mot « est ».

Le cœur du logiciel est composé de la classe “HashMap.cpp”, qui représente une table de hachage (<https://openclassrooms.com/courses/apprenez-a-programmer-en-c/les-tables-de-hachage>), c-à-d. une structure qui peut stocker des “mappings” d'une clef (ici un entier) à une valeur (ici un string). Par exemple, un HashMap avec les entrées (1,”log1000”) et (2,”TP1”) retournera la valeur “TP1” si on passe la clef “2”, tandis que rien ne sera retourné lorsque l'on donne la clef 3 (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'Équipier1 et Équipier2. Il vous faudra spécifier au début de votre rapport qui entre vous est réellement **l'Équipier1** et **l'Équipier2**. Par **exemple**, Équipier1 : **Khalil** et Équipier2 : **Mounia**. Au final, afin que le logiciel soit utilisable, il vous faudra développer un Makefile permettant de construire et déployer les outils.

1. Git

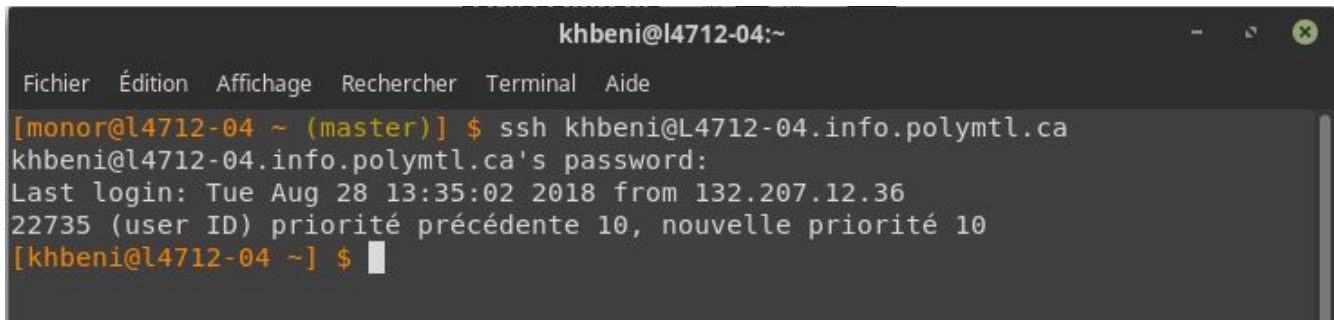
Préparation du poste de travail et documentation

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».

A screenshot of a terminal window titled 'khbeni@l4712-04:~'. The window has a menu bar with 'Fichier', 'Édition', 'Affichage', 'Rechercher', 'Terminal', and 'Aide'. The terminal shows a user 'monor@l4712-04 ~ (master)' running the command 'ssh khbeni@L4712-04.info.polymtl.ca'. The output shows the password prompt, the last login time 'Tue Aug 28 13:35:02 2018 from 132.207.12.36', and the user ID '22735 (user ID)'. The prompt then changes to '[khbeni@l4712-04 ~] \$'.

Remplacez «nomusager_polytechnique» par votre nom d'utilisateur 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 Linux2 pour obtenir un environnement similaire aux Tps. La documentation git proposée dans le cadre du cours explique très bien comment réaliser les différentes étapes de ce laboratoire. Elle 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 l'URL suivante : <https://github.gi.polymtl.ca/git/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 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'Équipier1). 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) prendre une capture d'écran de 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 log » et « git log -p » ?**

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** ». Chaque équipier crée une branche sur laquelle, il va faire des modifications pour ensuite la merger avec la branche principale. Vous devez donc réaliser les étapes suivantes :

1) Chaque équipe crée une branche avec la commande suivante:

```
$ git branch [nom de la nouvelle branche]
```

2) Chaque équipier doit basculer vers sa branche et pour le faire, il faut mettre la commande suivante:

\$ git checkout [nom de la nouvelle branche]

3) L'Équipier1 décommente la signature de la méthode `int compter (const std::string& key)` dans le fichier « `HashMap.h` » qui incrémente la valeur du key stocké dans le `HashMap` (si la clef existe) ou ajoute la clef avec valeur 1 (si la clef n'existait pas encore). il complète cette méthode en bas du fichier « `hashMap.cpp` » en utilisant les méthodes existantes de la classe. il Compilez manuellement le programme pour contrôler si le programme compile bien : « `g++ -o pari *.cpp` ». Ensuite exécutez le programme avec la commande `./pari`.

Suite à la modification, l'équipier doit exécuter la commande « `git status` ». Répondez par la suite aux questions suivantes :

a) prendre une capture d'écran de la sortie et collez à votre rapport la sortie du terminal correspondante à l'exécution de la commande.

b) Est-ce que cette situation est normale ? Pourquoi (pas) ?

4) Pour terminer, l'équipier en question exécute les commandes : « `git add .` » suivi de « `git commit -m "Votre Message"` » et « `git push --set-upstream origin [nom de la branche de l'équipier 1]` ». À ce point, une nouvelle révision de « `HashMap.h` » et « `HashMap.cpp` » devrait être créée.

Ensuite, l'équipier 1 doit faire un « `git merge master` », « `git checkout master` » et « `git merge [nom de la branche de l'équipier 1]` ».

5) L'Équipier2 doit par la suite, sous son répertoire et sur sa branche, 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.

Afin de propager sa modification, l'Équipier2 exécute les commandes : « `git add .` » suivi de « `git commit -m "Votre Message"` » et `git push --set-upstream origin [nom de la branche de l'Équipier 2]` » et une nouvelle révision de « `SomeKeyHash.cpp` » est ainsi créée.

Ensuite, l'équipier 2 doit faire un « `git merge master` », « `git checkout master` » et « `git merge [nom de la branche de l'équipier 2]` ».

Répondez par la suite aux questions suivantes :

a) prendre deux captures d'écran de la sortie du terminal correspondantes à l'exécution

de la commande « git merge » par chaque Équipier.

b) Est-ce que git a détecté un conflit ? Pourquoi (pas) ?

Sur votre terminal écrivez **GITK** “cette commande va vous permettre d’activer une fonctionnalité graphique du GIT”. après à l’aide de la vidéo suivante essayez de faire une comparaison entre vos deux branches. (<https://www.youtube.com/watch?v=PrgdepMiMLU>)
cette vidéo sera très utile pour le reste du TP.

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 repartiront leur travail.

1) L’Équipier1 et l’Équipier2 prennent le soin de faire, en tant que bonne pratique, un « git pull » avant de débiter leurs travaux sur le fichier en question.

2) L’Équipier1 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) Pourquoi le nom de l’exécutable, HashMap.h et HashMap.cpp sont écrits en rouge dans la console ?**
- b) prendre une capture d’écran de la sortie du terminal correspondante à l’exécution de la commande.**
- c) Compilez le code source et prendre une capture d’écran de la sortie du programme.**

4) En entretemps, l’Équipier2 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'Équipier1 exécute les mêmes actions « git add . » suivi de « git commit -m "Votre Message" » et « git push ».

Après, pour propager les modifications, l'Équipier2 exécute les mêmes actions que décrites précédemment en E 1.3 5). Répondez par la suite aux questions suivantes :

- a) **prendre une capture d'écran de la sortie du terminal correspondante à l'exécution des commandes «git fetch» et «git log --graph --decorate --glob="*" --oneline --name-only».**
- b) **Est-ce qu'il y aura un conflit lors d'un merge ? Pourquoi (pas) ? Si nécessaire, utilisez « git diff » pour décider.**
- c) **Maintenant faites « git merge », et faites une capture d'écran.**
- d) **Finalement, l'Équipier2 doit exécuter la commande « git push » suivi de « git log » et prendre une capture d'écran de la sortie 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 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'Équipier1 et L'Équipier2 prennent le soin de faire, en tant que bonne pratique, un « git pull » avant de débiter leurs travaux sur le fichier en question.
- 2) L'Équipier1 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 compter() 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'Équipier2 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'Équipier2 assume que l'Équipier1 est en train d'écrire le code source qui remplira « mymap ». Notez que l'Équipier2 n'enlève pas l'ancien code de `main()`.

- 5) Pour propager ses modifications, l'Équipier2 exécute les mêmes actions que décrites précédemment en E 1.2 3). Répondez aux questions suivantes :
- a) **prendre une capture d'écran de la sortie du terminal correspondante à l'exécution de la commande «git pull».**
 - b) **Est-ce que Git a détecté un conflit ? Pourquoi (pas)? Utilisez «git log --graph --decorate --glob="*" --oneline --name-only» et «git diff» pour supporter votre explication.**
 - c) **Dans le cas où vous rencontrez une situation conflictuelle, comment pensez-vous régler cette dernière si on veut que le logiciel résultant réussisse à résoudre le pari avec l'étudiant ? Est-ce que ça peut être fait automatiquement ? Pourquoi (pas) ? Si oui, résolvez le conflit.**
- 6) **Finalement, l'Équipier2 doit exécuter la commande « git push » suivi de « git log -v » et prendre une capture d'écran de la sortie correspondante au rapport sous la section «log ». S'il n'y a pas de différence entre ce « log » et celui de E1.3, veuillez consulter le chargé de laboratoire.**
- 7) [POINT BONUS] Quel est le mot le plus populaire dans « [The Art of Computer Programming](#) » ? La bonne réponse doit montrer la sortie de votre programme, incluant la fréquence du mot gagnant.
- 8) Veuillez décrire dans 3 lignes max l'utilité des branches dans GIT.

E2. 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'une 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 infra.txt examen.txt echo "Fini la première année!"
```

```
annee2: log2000.txt echo "Fini la deuxième année!"
```

```
#etc.
```

```
log1000.txt: infra.txt examen.txt cat infra.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 première année! Fini la deuxième année! ... Wouhou
```

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

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 **files/**, puis copie les fichiers générés par la compilation (de **build/**) ainsi que des fichiers .doc de **data/** (comme le livre « [The Art of Computer Programming](#), »5) vers le dossier **files/**.

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 .doc dans le dossier **data**, l'installation doit être refaite, mais sans réé-compilation du code source. Ce cas peut être testé avec la commande « touch data/hashage.txt make » dans le terminal.

a) **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. Ecrivez 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 votre graphe textuel.

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. Ne vous attardez pas à optimiser le script du **Makefile**. Suivez les étapes suivantes (astuce : ajoutez une commande "echo [un mot identifiant la règle]" dans chaque liste de commandes pour indiquer si les commandes de cette 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

E2.3 Un Makefile en JSON

ref : https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

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

1) Sortez du dossier que vous avez utilisé pour la partie 1, et créez un nouveau dossier dans lequel vous allez exécuter cette commande : `git clone https://github.com/nlohmann/json.git` Cela va vous permettre d'avoir une copie de la composante du logiciel json sur laquelle vous allez travailler dans le TP2. Ici on vise à compiler la composante.

- allez sur le dossier télécharger (JSON)
- suivez les étapes suivantes:

```
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

a) Maintenant exécutez la commande « `time make` » et insérez la capture de la sortie.

b) sur le chemin `/json/test/src` chercher le fichier `unit-comparison.cpp` remplacer `SECTION("values")` par la section suivante:

```
SECTION("values")
{
    json j_values =
    {
        nullptr, nullptr,
        -17, 42,
        8u, 13u,
        3.14159, 23.42,
        "foo", "bar",
        true, false, -117, 412,
```

```

81u, 131u,
31.114159, 213.412,
"foooo", "baoor",
true, false, -1347, 4342,
8u, 1334u,
3.14159, 23.42,
"foooo", "baoor",
true, false,
{1, 2, 3}, {"one", "two", "three"},
{{"first", 1}, {"second", 2}}, {"a", "A"}, {"b", {"B"}}
};

```

Maintenant exécutez la commande « **time make** » et insérez la capture de la sortie.

c) Finalement, changez de **true** à **false** la valeur du premier attribut dans la fonction **TEST_CASE("pointer access")** de la classe **unit-pointer_access.cpp** (/usagers/khbeni/json/test/src). Puis, exécutez une troisième fois la commande « **time make** » et insérez la capture de la sortie.

d) Est ce qu'il y a une différence de temps entre les trois exécutions de la commande ? Pourquoi (pas) ?

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://github.com/polymtl.ca/git/log1000-XX> » afin de voir les fichiers dans le dernier snapshot de 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 !!

21 septembre 2018 à minuit (23h55) POUR B1

14 septembre 2018 à minuit (23h55) POUR B2

Pénalités pour retard

10% par jour