

# Git

## Travail Pratique 1

---

Ce travail est à faire en équipe de 1, 2 ou 3 étudiants.

---

### Mise en contexte

*What I cannot create, I do not understand  
(Ce que je ne peux créer, je ne comprends pas).*  
- Richard Feynman

Utiliser un outil sans trop savoir comment il fonctionne peut parfois donner des résultats surprenants. Question de réduire les mauvaises surprises dans votre futur, votre premier travail consistera à implémenter un client *git* minimaliste nommé *gitus*.

---

### Spécification fonctionnelle

Cette section dénote l'utilisation générale de *gitus*.

#### Programme général

Premièrement, il est possible d'avoir de l'information de l'application avec le paramètre « --help ».

```
$ ./gitus -help
usage: gitus <command> [<args>]

These are common gitus commands used in various situations:

init      Create an empty Git repository or reinitialize an existing one
add       Add file contents to the index
commit    Record changes to the repository
```

Le programme que vous aurez à écrire devra offrir trois commandes (*init*, *add* et *commit*). Le détail de ces commandes est donné ci-dessous.

Notez qu'il n'est pas nécessaire que l'exécution de ces commandes soit transactionnelle. Il est acceptable qu'un échec laisse des traces sur le disque.

D'ailleurs, l'usage attendu de *gitus* par un client est le même que celui de l'outil ligne de commande *git*. En d'autres termes, à chaque exécution, *gitus* ne va exécuter qu'une seule commande. Une session d'utilisation pourrait donc ressembler à ceci:

```
$ ./gitus init
$ ./gitus add test.txt
$ ./gitus commit "Message" "Félix-Antoine Ouellet" uncourriel@usherbrooke.ca
```

De plus, toujours dans l'optique de reproduire le comportement de l'outil ligne de commande *git*, il devra fournir un guide d'utilisation lorsque utilisé avec l'argument « *--help* » ou lorsqu'appelé sans aucun argument. Ce guide devra être celui présenté ci-haut.

Une logique similaire devra être mise en place pour les commandes. Ainsi, l'invocation d'une commande avec l'argument « *--help* » devra produire un guide d'utilisation. Les sous-sections suivantes donnent les guides d'utilisation pour chaque commande.

#### Commande *init*

```
$ ./gitus init --help
usage: gitus init
```

La commande *init* devra initialiser un dépôt *git* dans le dossier courant c.-à-d. le dossier à partir duquel *gitus* est invoqué. Notez qu'il n'est demandé que de mettre en place les structures participantes activement à la gestion des sources. Toutes autres structures peuvent donc être absentes suite à l'exécution de cette commande.

Afin d'avoir une certaine simplicité dans ce travail, il n'y a seulement que le répertoire *.git*, le répertoire *.git/objects* et le fichier *.git/index* qui seront utilisés.

#### Commande *add*

```
$ ./gitus add --help
usage: gitus add <pathspec>
```

La commande *add* devra ajouter un fichier au *staging area* d'un dépôt *git*. Contrairement à la commande *add* de l'outil ligne de commande *git*, cette commande n'a pas à supporter l'ajout simultané de plusieurs fichiers.

Le paramètre « *pathspec* » devra correspondre à un chemin d'accès de fichier.

L'ajout au *staging area* se fait par l'insertion d'information dans le fichier *.git/index* et dans le répertoire *.git/objects* (sous forme de *blob*).

Le répertoire *.git/objects* doit être identique au traitement fait par *git*. La structure des informations contenues dans *.git/index* est à la discrétion des équipes (il est possible de ne pas respecter le standard *git*).

#### Commande *commit*

```
$ ./gitus commit --help  
usage: gitus commit <msg> <author> <email>
```

La commande *commit* devra ajouter produire un *changeset* contenant tout les fichiers ayant été ajouté à un dépôt *git*. En d'autres termes, elle devra produire un *changeset* avec le contenu du *staging area* d'un dépôt *git*.

Le paramètre « *msg* » devra correspondre au message du *changeset*. Le paramètre « *author* » devra correspondre au nom de l'auteur du *changeset*. Le paramètre « *author* » devra correspondre au courriel de l'auteur du *changeset*.

*Committer* en *git* est un processus en quatre étapes :

- 1- Le programme doit créer un arbre (*tree*) du répertoire central (*root*). Cet arbre contient l'information de tous les éléments présents dans ce répertoire. Bien sûr, il peut y avoir un autre répertoire. Dans ce cas, l'arbre du répertoire central doit contenir un arbre. Il est possible que cet arbre contienne un autre arbre s'il y a un autre répertoire à l'intérieur de ce répertoire (et ainsi de suite -- de manière récursive). Bien sûr, ceci s'applique seulement sur les éléments dans le *staging area*.
- 2- Le programme doit créer un *commit*. La structure d'un *commit* doit être identique à celle de *git* (auteur, *committer*, courriel, l'heure le *commit* parent (si présent) et le message « *msg* »).
- 3- Le programme doit éliminer les changements fait dans le fichier *.git/index*. Il est possible de conserver quelques informations entre les *commits* (un peu comme *git* fait).
- 4- Le programme doit changer la tête de la branche. Ceci n'est nécessaire à faire pour ce travail (il n'y aura aucune pénalisation si cette étape n'est pas implémentée).

Commandes bonus : *branch* et *checkout*

```
$ ./gitus branch --help
usage: gitus branch
usage: gitus branch <branchName>

$ ./gitus checkout --help
usage: gitus checkout <branchName>
```

La commande *branch* a deux directives:

- Afficher toutes les branches disponibles;
- Créer une branche avec le nom « *branchName* »

Afin d'accomplir cette tâche, il est nécessaire de modifier les commandes *init*, *add* et *commit*.

Attention: l'accomplissement de cette section nécessite plus de temps que le nombre d'heures alloué pour ce travail.

---

Remise

La remise de ce travail devra être faite avant 28 janvier 2020 à 6h00. Aucun retard ne sera accepté.

Tout le contenu de ce travail doit se retrouver dans une branche nommée *TP1*. Cette branche ne doit pas être intégrée dans la branche maîtresse (*master*). Le dossier à la racine doit se nommer *gitus* se situant à la racine de votre dépôt. Un manque à cette directive entraînera une note de 0%.

Pour remettre ce travail, vous devrez apposer une étiquette (*tag*) portant le nom *TP1* sur le *commit* final de votre travail. Un manque à cette directive entraînera une note de 0%.

Prenez note que ce travail sera corrigé sur une machine roulant Ubuntu 18.04. Les compilateurs présents sur cette machine sont *GCC* (<https://gcc.gnu.org/>) (version 8.2) et *Clang* (<https://clang.llvm.org/>) (version 8). Vous pouvez aussi compter sur la présence de la bibliothèque *Boost* (<https://www.boost.org/>) (version 1.70) ainsi que toutes bibliothèques nécessaires au bon fonctionnement de toutes les sous-bibliothèques de *Boost*. Un travail qui ne compilerait dans cet environnement recevra automatiquement une note de 0%.

Pour tout message dédié au correcteur, veuillez ajouter un fichier *README* à la racine de votre dépôt (et sur la bonne branche). Minimale, le fichier devrait contenir les auteurs du travail et le travail réalisé (les commandes fonctionnelles).

De plus, afin d'aider la correction, il serait gentil d'indiquer si le bonus a été implémenté.

## Détails techniques

Dans le but de ne pas vous encombrer avec des notions de gestion technique de projet qui n'ont pas encore été abordées en classe, un projet de base vous est fourni.

Par souci de portabilité, ce projet utilise le métasystème de production *CMake* (<https://cmake.org/>). Il est recommandé de suivre cet excellent tutoriel (<https://preshing.com/20170511/how-to-build-a-cmake-based-project/>) pour comprendre comment utiliser *CMake*. Plusieurs exemples sur plusieurs plateformes y sont présentés.

Finalement, notez que vous devrez installer manuellement les bibliothèques *Boost* et *ZLib* sur votre poste de travail pour pouvoir compiler ce projet.

Autres détails :

- Le travail doit être écrit en C++.
- Les bibliothèques permises sont:
  - La bibliothèque standard de C++ (<https://fr.cppreference.com/w/cpp/header>)
  - La bibliothèque Boost (<https://www.boost.org/>)
  - La bibliothèque Catch2 (<https://github.com/catchorg/Catch2>)
  - La bibliothèque ZLib (<https://zlib.net/>)
- Outre qu'il répond aux exigences formulées dans la section précédente, il est attendu que votre code soit:
  - Robuste
  - Maintenable
  - Efficace
  - Portable
  - Moderne
- Le code remis devra être testé à l'aide de tests programmés. Ces tests devront mettre à contribution *Catch2* (<https://github.com/catchorg/Catch2>).

---

## Tests

Vous devez implémenter des tests à l'aide de la bibliothèque *Catch2* (<https://github.com/catchorg/Catch2>). Il n'est pas nécessaire d'installer la librairie; elle est fournie dans le projet de base et ne requiert qu'une seule inclusion. Dans le projet de base, il y a déjà un exécutable de tests qui est généré. Vous n'avez seulement qu'à implémenter les différents tests.

Afin d'apprendre à vous servir de cette librairie, il est fortement conseillé de lire la documentation disponible sur le site web de *Catch2*.

Il est attendu d'avoir des tests des commandes qui couvrent tous les cas possibles. Pour faciliter votre travail, vous pouvez implémenter 3 fonctions (une pour chaque commande) et testez ces fonctions (de manière unitaire). Ces fonctions peuvent valider les paramètres à l'intérieur (ceci va grandement faciliter le travail).

---

## Grille de correction

Voici le maximum de points possible par consignes:

- *init* 10 points
- *add* 20 points
- *commit* 35 points
- *branch* (et *checkout*) 10 points
- *tests* 25 points

## Pénalités potentielles

Cette section représente la grille de correction à proprement parler. Elle liste donc toutes les pénalités qu'un travail peut se mériter. Par souci de clarté, ces pénalités sont regroupées selon les mêmes catégories que celles indiquées dans l'énoncé à la section « Détails techniques ». De plus, chaque pénalité, excepté celles des sections « Détails techniques » et « Tests », est accompagnée d'un indicatif correspondant au sigle du cours dans lequel une notion donnée aurait dû être maîtrisée ou, du moins, bien comprise et assimilée par les étudiants. Autant que possible, il y a un effort de donner des exemples de manquement. Ces exemples ne sont cependant pas exhaustifs, car déterminer d'avance toutes les erreurs que les étudiants peuvent faire relève de la prescience. Finalement, il est à noter que la pénalité associée à un manquement peut être vue comme étant le maximum de points déductibles. Les déductions peuvent être moindres selon la sévérité d'un manquement ainsi que son nombre d'occurrences. Cette décision est laissée au bon jugement du correcteur.

Commented [AB1]: A transférer dans un nouveau fichier

## Respect des consignes

- Le code remis ne compile pas dans l'environnement de correction:-100%
- Travail est remis en retard:
  - Retard de moins de 24 heures:-50%
  - Retard de plus de 24 heures:-100%
- Remise non-conforme:-100%
  - *Branche, tag, ...*
- Lecture du fichier d'informations non-implémentée:-10%
- Messages d'erreurs non descriptifs:-5%
- Dédoublage de code inutile :-10%
- Commandes non implémentées : tous les points de la commande
- Manquement à l'interface d'une commande : jusqu'à 30% des points de la commande

## Tests

- Cas à succès non-testés ou tests trop en surface (liste minimale des éléments à tester ci-dessous):-25%
  - Bon fonctionnement des commandes (*add, init, commit*)
  - Vérification de l'état des fichiers du *.git*
  - Vérification des erreurs
  - Il n'est pas nécessaire de tester les écritures à l'écran

## Robustesse

- Utilisation de variables globales mutables:-5% (**IFT232**)
- Manque de *const-correctness* au niveau des méthodes: -5% (**IFT339**)
- Manque de distinction entre variables et constantes: -5% (**IFT159**)
- Mauvaise gestion d'erreur:-5% (**IFT339, IFT232**)
  - Exemple mineur: Échouer silencieusement sans rapporter d'erreur au client (client peut référer autant à un être humain qu'à une fonction appelante selon le contexte)
  - Exemple majeur: Laisser fuir une exception dans l'application (va faire planter le programme)
- Manque de validation de pré-conditions:-5% (**IFT232**)
  - Exemple: Ne pas valider un pointeur avant de le déréférencer

- Manque de validation de post-conditions:-5% (**IFT232**)
- Variables membres non-initialisées:-5% (**IFT339, IFT232**)
- Utilisation de variables locales non-initialisées:-5% (**IFT159, IFT339**)
- Prise de référence menant à une *dangling reference*:-5% (**IFT339**)
  - Exemple : Prise de référence sur une variable locale d'une portée moindre

#### Maintenabilité

- Fonction effectuant plusieurs actions disjointes :-5% (**IFT159**)
- Sur-encapsulation:-5% (**IFT232**)
  - Exemple: Fonction membre n'utilisant aucune variable membre
- Sous-encapsulation (classes):-5% (**IFT232**)
  - Exemple: Classe exposant ses variables membres
- Sous-encapsulation (modules):-5% (**IFT232**)
  - Exemple: Exposition de fonctions utilitaires dont l'usage devrait être uniquement interne au module
- Utilisation d'héritage où la composition aurait été préférable:-5% (**IFT232**)

#### Efficacité

- Utilisation de sémantique de valeur où la sémantique de référence aurait été préférable pour passer des paramètres de fonctions:-5% (**IFT339**)
- Copies inutiles:-5% (**IFT339**)

#### Portabilité

- Usage de bibliothèques non-portables:-5% (**IGL601**)
  - Exemple: utiliser des inclusions de `<windows.h>`
- Usage d'extensions de compilateur non-portables:-5% (**IGL601**)
  - Exemple: utiliser l'extension `__super` qui n'est disponible qu'avec le compilateur `MSVC`
- Usage de types dont la taille peut varier alors que la situation demande une taille fixe:-5% (**IGL601**)
  - Exemple: utiliser `size_t` au lieu de `uint32_t` ou `uint64_t`

#### Pratiques de programmation

- Absence de commentaires là où il aurait été pertinent d'en mettre:-5% (**IFT159**)
- Mauvaise indentation rendant la lecture du code difficile:-5% (**IFT159**)



- Noms non-significatifs:-5% (**IFT159**)
  - Exemple classique: Nommer la variable d'induction d'une boucle *for* tout simplement *i*
- Mauvaise utilisation d'une structure de contrôle:-10% (**IFT159**)
  - Exemple 1: *switch* sur une variable booléenne
  - Exemple 2: *goto* en général
- Code mort:-5% (**IFT159**)
  - Exemple 1: Fonction jamais appelée
  - Exemple 2: Code après un énoncé *return* incondionnel.
- Architecture difficile à comprendre : 20% (**IFT159, IFT232**)

## Modernité

- Ré-implémentation de fonctionnalités déjà offertes par bibliothèque standard:-5% (**IFT159, IFT339, IFT232, IGL601**)
  - Exemple: Réimplémenter une fonction se trouvant dans le *header* standard `<algorithm>` tel que `std::sort`.
- Utilisation de pointeurs bruts pour signifier la possession:-5% (**IFT339**)
- Pollution du *namespace* global:-5% (**IGL601**)
  - Exemple: Utilisation de `using namespace std;`

---

## Références utiles

- <https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain> (surtout pour la 2e page)
- <https://pragmaticjoe.blogspot.com/2015/02/how-to-generate-sha1-hash-in-c.html>
- <https://stackoverflow.com/questions/29217859/what-is-the-git-folder>
- <https://codewords.recurse.com/issues/two/git-from-the-inside-out>
- <https://mincong-h.github.io/2018/04/28/git-index/>
- [https://matthew-brett.github.io/curious-git/reading\\_git\\_objects.html](https://matthew-brett.github.io/curious-git/reading_git_objects.html)
- [https://www.boost.org/doc/libs/1\\_71\\_0/libs/iostreams/doc/home.html](https://www.boost.org/doc/libs/1_71_0/libs/iostreams/doc/home.html)
- <https://zlib.net/>
- <https://stackoverflow.com/questions/7282645/how-to-build-boost-iostreams-with-gzip-and-bzip2-support-on-windows>