Date de présentation : 12 janvier 2022 Auteur : Félix-Antoine Ouellet, Alex Boulanger
Date de remise : 1er février 2021 Correctrice : sarah.dupont@usherbrooke.ca

Gitus

Travail Pratique 1

Ce travail est à faire en équipe de 1 ou 2 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*.

Description

Gitus est une application console qui opère sur une base de données locale et propriétaire. Lors d'une exécution, *gitus* s'ouvre, exécute la commande, interagit avec la base de données et se ferme. Lors d'une seconde opération (ainsi que toutes autres subséquentes), *gitus* continue d'interagir avec la même base de données.

Spécification fonctionnelle

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

Programme général

Il est possible d'avoir de l'information de l'application avec le paramètre « --help ». (Les explications en français sont également possibles)

```
$ ./ 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"
```

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 typiques de *git* 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*, le fichier *.git/index* et le fichier *.git/HEAD* 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. Si le paramètre n'est pas spécifié, l'opération doit échouer.

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*. Les noms générés des fichiers à insérer dans *.git/objects* par gitus doivent être identique à ceux générer par l'application de git. Cependant, les contenus de ces fichiers vont être différent du résultat de *git*; ceci est pour faciliter la correction. Il est demandé d'avoir, comme contenu, le nom du fichier, la taille du fichier et le contenu verbatim du fichier ajouté[1].

La structure des informations contenues dans *.git/index* est à la discrétion des équipes (ne respectez pas le standard *git*). Le contenu du fichier ne doit pas être encrypté et doit lisible par le correcteur.

Commande *commit*

```
$ ./gitus commit --help

usage: gitus commit <msg> <author>
```

La commande *commit* devra ajouter et produire un *changeset* contenant tous 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*. L'opération devra être invalide si un des deux paramètres est vide.

Afin d'avoir plus qu'un seul mot comme message, l'application doit supporter les guillemets « " » pour la délimitation. La même distinction doit être fait pour l'auteur.

Commiter en *gitus* est un processus en cinq étapes :

1- Le programme doit lire les éléments du fichier *.git/index;*
2- Le programme doit créer un arbre (*tree*) à partir du répertoire central (*root*) et l'écrire dans la base de données. Cet arbre contient l'information de tous les éléments présents dans ce répertoire et des sous-répertoires[2]. Bien sûr, ceci s'applique seulement sur les éléments dans le *staging area* et les éléments déjà présents dans le répertoire. Pour

[1] La délimitation peut être des sauts de lignes, des espaces ou tout autres
[2] Contairement à Git. Fait pour simplification du travail.

connaitre les éléments présents dans le répertoire, il faut consulter l'arbre du précédent *commit*. L'application doit également générer un sha1 pour l'arbre;

3- Le programme doit créer un *commit* et l'écrire dans la base de données. Les informations sauvegardées doivent être l'auteur, le *commit* parent (si présent), l'arbre (son sha1), la date (incluant l'heure, minutes et secondes)[3] et le message « *msg* ». Le fichier d'information de *commit* doit avoir un sha1 généré similairement que celui généré dans le *add* (sans être obligé de suivre la convention de *git*);
4- Le programme doit éliminer les changements faits dans le fichier *.git/index*;
5- Le programme doit connaitre quel est le plus récent commit. Il faut donc insérer le sha1 du *commit* dans le fichier .git/HEAD.

Checkout (Bonus)

```
$ ./gitus checkout --help

usage: gitus checkout <commitID>
```

La commande *checkout* doit changer les fichiers présents pour les versions des fichiers utilisés avec la révision « *commitID* » où le *commitID* est le sha1 d'un *commit*.

Vous devez bloquer toute tentative de soumissions si le dépôt n'est pas à la dernière soumission.

Remise

La remise de ce travail devra être faite avant 30 mai 2021 à 23h59. 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 maitresse (*master*). Le dossier à la racine doit se nommer *gitus* et se situer à la racine de votre dépôt. Un manque à cette directive entraînera une perte de 20%.

Pour remettre ce travail, vous devrez apposer une étiquette (*tag*) portant le nom *TP1-tag* sur le *commit* final de votre travail. Un manque à cette directive entraînera une perte de 20%.

Prenez note que ce travail sera corrigé sur une machine créer avec les mêmes spécifications que le laboratoire 1. Un travail qui ne compilerait dans cet environnement recevra une perte de 50%.

[3] std::chrono

Votre dépôt doit avoir obligatoirement un README. Minimalement, le fichier devrait contenir les auteurs du travail et le travail réalisé (les commandes fonctionnelles). Un fichier n'incluant pas les auteurs ni message ne sera pas corrigé. Ce fichier sera lu lors de la correction et est l'endroit idéal pour tout message au correcteur.

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 remplacer et 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).

Astuces

Ne tentez pas de régler des problèmes imaginaires. Concentrez-vous sur les problèmes actuels. Par exemple, ce travail comporte 3 (ou 4 instructions); l'implémentation du « Command Design Pattern » est jugée comme superflus et inutile…

Conserver la solution simple et facile pour la correction.

Grille de correction

Voici le nombre de points par consignes (excluant les autres pénalités) :

- *init* 15 points
- *add* 25 points
- *commit* 35 points
- *checkout* (Bonus) 10 points
- *tests* 25 points
  - *init* 5 points
  - *add* 10 points
  - *commit* 10 points
  - *checkout* (bonus) 5 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.

Respect des consignes

- Le code remis ne compile pas dans l'environnement de correction:-50%
- Travail est remis en retard :-100%
- Remise non-conforme:-20%+
  - *Branche, tag, …*
- Messages d'erreurs non descriptifs:-5%
- Dédoublement de code inutile :-5% par instance
- 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):
  - Bon fonctionnement des commandes (*add*, *init*, *commit*)
  - Vérification de l'état des fichier du *.git*
  - Vérification des erreurs
  - Il n'est pas nécessaire de tester les écritures à l'écran

Robustesse – par occurrence

- 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 – par occurrence

- 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)***
- Solution trop complexe pour le problème : -35% (IFT159, IFT232)

Efficacité – par occurrence

- 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 – par occurence

- 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 – par occurrence

- Absence de commentaires là où il aurait été pertinent d'en mettre:-5% ***(IFT159)***
- Mauvaise indentation rendant la lecture du code difficile:-2 à-5% ***(IFT159)***

- Noms non-significatifs:-2% ***(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* inconditionnel.
- Architecture difficile à comprendre :-20% ***(IFT159, IFT232)***

Modernité – par occurrence

- 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://www.boost.org/doc/libs/1_73_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