Auteur et correcteur: Félix-Antoine Ouellet (felix-antoine.ouellet@usherbrooke.ca)

Enseignant: Alex Boulanger (alex.boulanger@usherbrooke.ca)

Devoir 1

Ce travail est à faire en équipe de 2 ou 3 étudiants. Si vous ne parvenez pas à vous trouvez une équipe, contactez l'enseignant et le correcteur.

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

Programme général

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

Cependant, en toutes circonstances, l'utilisateur devra être averti d'un échec avec le même message que l'outil ligne de commande *git* produirait pour le cas donné.

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" ouef2901@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 lorsque 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 participants activement à la gestion des sources. Toutes autres structures peuvent donc être absentes suite à l'exécution de votre programme.

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 paramêtre *pathspec* devra correspondre à un chemin d'accès de fichier.

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

Spécification technique

- Le travail doit être écrit en C++.
- Les bibliothèques permises sont:
- La bibliothèque standard de C++
- La bibliothèque Boost
- Outre qu'il réponde aux exigences formulés dans la section précédente, il est attendu que votre code soit:
- Robuste
- Maintenable
- Efficace
- Portable
- Moderne

- Le code remis devra être testé. Ces tests devront mettre à contribution Boost.Test.

Projet de base

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

Par souci de portabilité, ce projet utilise le méta-système de production CMake. Il est recommandé de suivre cet excellent tutoriel pour comprendre comment utiliser CMake. Plusieurs exemples sur plusieurs plateformes y sont présentés.

Finalement, notez que vous devrez installer manuellement la bibliothèque Boost sur votre poste de travail pour pouvoir compiler ce projet.

Remise

La remise de ce travail devra être faite avant 18 mai 2019 à 6h00. Une pénalité de 50% sera accordé pour une remise dans les 24 heures suivantse. Un travail remis plus de 24 heures en retard recevra la note de 0%.

Tout le contenu de ce travail doit se retrouver dans un dossier nommer **TP1** 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 (version 8.2) et Clang (version 8). Vous pouvez aussi compter sur la présence de la bibliothèque Boost (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%.

En guise de grille de correction

Une particularité de la correction dans ce cours est qu'elle est modelée sur les revues de code faites en entreprise. Le travail remis devra être donc être de qualité sous plusieurs angles. Vous trouverez ci-dessous une liste d'éléments communs pouvant être pénalisés dans vos travaux lors de la correction. Notez que cette liste n'est pas totalement exhaustive. Dans le doute, consultez le correcteur ou l'enseignant.

Respect des consignes

- *init* non-implémenté
- *init* ne respecte pas la spécification
- *add* non-implémenté
- *add* ne respecte pas la spécification
- *commit* non-implémenté
- *commit* ne respecte pas la spécification
- Message d'erreur différent de celui de l'outil ligne de commande *git* pour une situation donnée
- La structure du répertoire *.git* produit par *gitus* diffère de ce que l'outil ligne de commande *git* produit (uniquement pour les éléments contribuant à la gestion des données).
- Utilisation de bibliothèque(s) autres que celles permises.

Tests

- Cas à succès non-testés
- Test non-reproduisible

Robustesse

- Utilisation de variables globales mutables

- Manque de *const-correctness*
- Mauvaise gestion d'erreur
- 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 une fonction qualifiée *noexcept* (cause un crash)
- Manque de validation de pré-conditions
- Exemple: Ne pas valider un pointeur avant de le déréférencer
- Manque de validation de post-conditions
- Variables membres non-initialisées
- Utilisation de variables locales non-initialisées

Maintenabilité

- Sur-encapsulation
- Exemple: Fonction membre n'utilisant aucune variable membre
- Sous-encapsulation (classes)
- Exemple: Classe exposant ses variables membres
- Sous-encapsulation (modules)
- Exemple: Exposition de fonctions utilitaires dont l'usage devrait être uniqument interne au module
- Utilisation d'héritage où la composition aurait été préférable

Efficacité

- Utilisation de sémantique de valeur où la sémantique de référence aurait été préférable.
- Code mort
- Exemple: Fonction jamais appelée
- Code invariant dans une répétitive
- Exemple: Ci-dessous *monVec.size()* devrait être capturé dans une variable constante avant le début de la boucle

```
std::vector<int> monVec;
/* ... */
for (int iVec = 0; iVec < monVec.size(); ++iVec)
{
std::cout << iVec << "/" << monVec.size() << " = " << monVec[i] << "\n";
}
```

Portabilité

- Usage de bibliothèques non-portables
- Ex: utiliser des trucs de <windows.h>
- Usage d'extensions de compilateur non-portables
- Ex: utiliser l'extension *___super* qui n'est disponible qu'avec *MSVC*
- Usage de types dont la taille peut varier alors que la situation demande une taille fixe
- Ex: 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
- Mauvaise indentation rendant la lecture du code difficile
- Noms non-significatifs
- Exemple classique: Nommer la variable d'induction d'une boucle *for* tout simplement *i*
- Mauvaise utilisation d'une structure de contrôle
- Exemple 1: *switch* sur une variable booléenne
- Exmeple 2: *goto* en général

C++ moderne

- Réimplémentation de fonctionnalités de la bibliothèque standard
- Exemple: Réimplémenter une fonction se trouvant dans le *header* standard <algorithm> tel que *std::sort*.
- Utilisation de pointeurs bruts pour signifier la possession.
- Pollution du *namespace* global
- Exemple: Utilisation de *using namespace std;*