Pablo Rauzy
pablo.rauzy@univ-paris8.fr
https://pablo.rauzy.name/teaching/ddll

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Développement de logiciels libres

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / L3 informatique

Séance 3 (TP) : Premiers pas avec Git

N'oubliez pas :

- Les TPs doivent être rendus par courriel au plus tard le lendemain du jour où ils ont lieu avec "[ddll]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[ddll] TP3 Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tp3").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net !**

Exercice 0.

Initialisation d'un dépôt local.

1. Avant de commencer à utiliser un répertoire comme dépôt, il doit être initialisé. Pour cela il suffit de taper la commande `git init` dans le répertoire.
   → Initialisez un dépôt git dans un répertoire vide.
2. → Quels sont les résultats de la commande ? Que peut-on voir dans le répertoire ?
3. On peut utiliser la commande `git status` pour voir l'état d'un dépôt Git.
   → Dans quel état est votre dépôt ?

Exercice 1.

Ajout de contenu et premiers commits.

1. Créez un fichier `main.c` dans votre dépôt avec ce contenu :

```c
#include <stdio.h>

int
main (int argc, char *argv[])
{
  printf("Hello, world!\n");
  return 0;
}
```

   → Une fois le fichier `main.c` créé, comment a changé l'état de votre dépôt ?
2. Pour suivre un fichier, on utilise la commande `git add` *`fichier`*.
   → Suivez le fichier `main.c` dans votre dépôt. Quel est maintenant l'état de votre dépôt ?
3. Pour valider des modifications, on utilise la commande `git commit`. On peut soit lui passer un message pour décrire les modifications avec l'argument `-m` (par exemple `git commit -m "mon premier commit"`) soit ne rien faire et dans ce cas Git ouvrira un éditeur pour que vous tapiez un message.
   → Validez l'ajout du fichier `main.c` dans votre dépôt. Que constatez-vous ?
4. → Pour configurer Git et lui dire quel est votre nom et votre adresse email, utilisez les commandes suivantes :
   `git config --global user.name "Votre Nom"`
   `git config --global user.email "votre@email"`.
5. → Que est l'état de votre dépôt Git maintenant ?
6. Créez rapidement deux fichiers textes `foo.txt` et `bar.txt` (avec une dizaine de lignes de quelques mots dans chacun, mais n'y passez pas de temps).
   → Quel est l'état de votre dépôt git maintenant ?

7. → Suivez les fichiers textes d'un seul coup avec la commande `git add *.txt`.
8. → Validez le suivi des deux fichiers textes avec `git commit -m "ajouts des fichiers textes"`.
9. Compilez le programme avec `gcc main.c` pour créer l'exécutable `a.out`.
   → Quel est l'état de votre dépôt git ?
10. Dans le dépôt, on veut suivre les versions des fichiers sources mais pas de l'exécutable.
    → Pourquoi ?
11. Git permet de lui donner une liste de fichiers à ignorer, il faut pour cela mettre leur nom (par exemple `a.out`) ou motif (par exemple `*.out`) dans un fichier appelé `.gitignore` à la racine du dépôt.
    → Dites à Git d'ignorer `a.out` en créant un fichier `.gitignore`, puis vérifiez l'état de votre dépôt pour voir si cela fonctionne bien.
12. → Dites maintenant à Git de suivre le fichier `.gitignore`, puis validez cette modification.

Exercice 2.

Modifications.

1. Modifiez un peu deux ou trois lignes vers le début du fichier `foo.txt` et ajoutez lui quelques lignes à la fin. Ensuite remplacez le "Hello, world !" dans le fichier C par "Bonjour tout le monde !".
   → Dans quel état est votre dépôt Git ?
2. Git vous permet de visualiser les modifications entre l'état actuel des fichiers dans le dossier et celui que lui a enregistré (soit validés, soit déjà dans la zone "à valider"). On utilise pour cela la commande `git diff`.
   → Visualisez les modifications que vous avez faites à la question précédente.
3. Comme vous le voyez, les modifications sont présentées ligne par ligne car c'est souvent le plus pratique avec du code. Il est tout de même possible de lui demander de visualiser les modifications mot par mot avec l'option `--color-words`.
   → Visualisez mot par mot les modifications que vous avez faites à la question 1.
4. Il est important quand on gère du code source de ne pas se servir du gestionnaire de version comme d'un simple engin de sauvegarde. On dit au contraire que l'historique du code doit être *sémantique*, c'est à dire que les commits doivent avoir un sens :
   — "Correction du bug #42" est un bon message de commit.
   — "Mon travail du week-end" n'est pas un bon message de commit.
   On va donc valider nos changements en deux fois.
   Pour mettre à jour ce qui sera validé, on utilise aussi `git add`.
   → Dites à git de mettre dans la zone de validation le fichier `main.c`. Quel est le nouvel état du dépôt ?
5. → Validez la modification du fichier `main.c` avec le message "traduction en français".
6. → Mettez le fichier `foo.txt` dans la zone de validation avec `git add`.
7. Oups, en fait non, on ne veut pas le mettre d'un seul coup ! Git nous permet de retirer un fichier de la zone de validation en utilisant `git reset` *`fichier`*.
   → Retirez le fichier `foo.txt` de la zone de validation.
8. Parfois on a fait plusieurs modifications pour des raisons différentes au même fichier sans les avoir validées au fur et à mesure. On va imaginer que c'est le cas de notre fichier `foo.txt` : les quelques lignes modifiées vers le début du fichier le sont pour corriger le bug #42 alors que les quelques lignes ajoutées à la fin du fichier implémentent la *feature request* #13.
   On va donc devoir faire deux commits. Git nous permet cela grâce à la commande `git add -p`, qui vous proposera quelles modifications vous voulez ou non ajouter dans la zone de validation.
   → Ajoutez les modifications qui concernent le bug #42 dans la zone de validation, puis vérifiez l'état du dépôt.
9. Avant de valider les modifications qui concernent le bug #42, vous pouvez vérifier que les modifications restantes correspondent bien seulement à l'implémentation de la feature request #13 avec `git diff`.
   → Si tout va bien, validez les modifications avec un message de commit approprié.
10. Comme on maintenant sûr qu'il ne reste que les modifications qui correspondent à l'implémentation de la feature request #13, on peut les ajouter d'un seul coupà la zone de validation puis les valider avec la commande `git commit -a -m "`*`message`*`"`.
    → Validez ces modifications avec un message de commit approprié.

Exercice 3.

Historique.

1. Pour visualiser l'historique de votre dépôt, il faut utiliser la commande `git log`.
   → Décrivez les informations que vous voyez dans l'historique.

2. On a parfois besoin de regarder rapidement l'historique, on peut le faire avec l'argument `--oneline`.
   → Quelle différence avec et sans cet argument ?
3. On peut aussi vouloir limiter le nombre de commits affichés, on peut le faire avec l'argument `-`n (en remplaçant n par le nombre souhaité de commits).
   → Affichez seulement le dernier commit.
4. Oups !! On a oublié dans le dernier commit de mettre une modification dans le fichier `bar.txt` dans lequel vous devez rajouter une ligne disant "feature #13 ok" à la fichier du fichier.
   Heureusement, Git nous permet de modifier le dernier commit quand on est tête en l'air, avec `git commit --amend`.
   → Ajoutez la ligne dans `bar.txt`, puis ajoutez ce fichier dans la zone de validation, pour enfin le rajouter au dernier commit, qui concerne la feature request #13.
5. On pourrait aussi imaginer que vous avez besoin de revoir la version anglaise du programme `main.c`.
   → Trouvez le nom du commit précédent celui où vous l'avez traduit en français.
6. On peut remettre le fichier dans l'état où il était à ce commit grâce à la commande `git checkout` *`commit fichier`*.
   → Remettez le fichier `main.c` en version anglaise. Quel est l'état du dépôt Git ?
7. Compilez et exécutez `main.c`. C'est bon, vous avez pu voir ce que vous vouliez voir, on va maintenant revenir dans l'état "normal". Il y a un alias pour le dernier commit de la branche courante, c'est `HEAD`.
   → Revenez à l'état actuel du fichier avec `git checkout HEAD main.c`. Quel est alors l'état du dépôt Git ?

Exercice 4.

Travail collaboratif.

1. Un des principaux points forts de git est de faciliter le travail collaboratif.
   Les dépôts distants sont appelés "remote".
   Généralement, on crée un dépôt "bare" qui va juste servir à synchroniser le travail des différents membres du projet.
   → Par groupe, venez me demander de créer un dépôt "bare" pour vous sur la VM `192.168.3.161`.
2. Sur la machine locale, on va ajouter le dépôt distant avec la commande `git remote add` *`nom adresse`*.
   Par convention on utilise généralement le nom `origin`, et ici l'adresse serait `git@192.168.3.161:`*`foo`*`.git` (où *`foo`* est le nom du dépôt que je vous aurais créé).
   → Toujours en groupe, sur l'une de vos machines locales, ajoutez le dépôt bare distant que vous venez de créer.
3. → Sur cette même machine, vous allez pousser l'historique local vers le dépôt distant avec la commande `git push origin master`.
4. Les autres membres du projet doivent maintenant cloner ce dépôt pour pouvoir travailler en collaboration, cela se fait avec la commande `git clone`.
   Attention, cela doit être fait *en dehors de votre dépôt* car `git clone` en crée un nouveau.
   → Sur les machines locales des autres membres du groupe, faites `git clone git@192.168.3.161:`*`foo`*`.git`.
5. Regardons maintenant à quoi ressemble le nouveau dépôt.
   → Quel est l'état de ce dépôt ? Que contient son historique ?
6. Chaque membre du group crée un nouveau fichier à son nom, le fait suivre par le dépôt puis valide cette modification.
   → Pour partager vos modifications, vous allez maintenant utiliser `git push`.
7. C'est toujours mieux avant de pousser ses propres modifications de mettre à jour le contenu de son dépôt local.
   → Pour récupérer les modifications des autres, utilisez `git pull`. Regardez l'évolution de l'historique de votre dépôt.
8. Mettez vous d'accord pour créer un conflit, par exemple deux membres du groupe essayent de modifier le même fichier.
   → Comment Git vous signale le conflit ? Ouvrez le fichier concerné, que constatez-vous ?
9. → Éditez le fichier pour le remettre dans le bon état puis validez les modifications pour résoudre le conflit.

Exercice 5.

Branches.

1. Git permet très facilement de créer des *branches*. On peut les voir comme des univers parallèles où le code évolue de façon différente.
   C'est par exemple une bonne pratique de garder sa branche master (celle par défaut) propre, et de faire son développement dans d'autre branche pour ne pas que le code dans la branche master se retrouve dans un état instable.
   → Listez les branches existantes dans votre dépôt avec `git branch`.

2. Créé une nouvelle branche se fait avec la commande `git branch` *nom*.

   → On veut maintenant rajouter une fonctionnalité à notre programme. Plutôt que de lui faire dire “Bonjour tout le monde !”, on veut qu’il salue la personne passée en argument. On va dire que c’est notre feature request #14.

   → Créez une branche “*votrenom*-fr14”. Quel est l’état de votre dépôt git ?

3. → Basculez sur la branche que vous venez de créer avec `git checkout` *votrenom*`-fr14`

4. On va commencer à implémenter la feature.

   (a) Modifiez le fichier `main.c` pour que la ligne 6 soit :

   ```
   1  printf("Bonjour %s !\n", argv[1]);
   ```

   (b) Compilez le avec `gcc main.c`.

   (c) Testez le avec `./a.out Sam`.

   → Validez la modification avec `git commit main.c -m "salut personnalisé"`.

5. Maintenant, on ne va pas simplement dire qu’on a fini car le programme pourrait planter dans l’état actuel, par exemple si on ne lui passe pas d’argument.

   (a) Modifiez le fichier `main.c` pour avoir dans le main :

   ```
   1  if (argc != 2) {
   2    fprintf(stderr, "Usage: %s <nom>\n", argv[0]);
   3    return 1;
   4  }
   5  printf("Bonjour %s !\n", argv[1]);
   6  return 0;
   ```

   (b) Compilez le avec `gcc main.c`.

   (c) Testez le avec `./a.out Sam` et `./a.out`.

   → Validez la modification avec `git commit main.c -m "gestion erreur nom pas fourni"`.

6. Maintenant on estime que le programme est prêt, alors vous poussez votre branche pour qu’elle soit vu par les autres membres du projet et qu’illes vous donne leur avis avant de la mettre dans master.

   → Poussez votre branche de développement avec `git push origin` *votrenom*`-fr14`.

7. Vos camarades voient le code et sont d’accord qu’il est bon, mais l’un·e d’entre elleux se plaint tout de même du cassage de la rétrocompatibilité avec les versions précédentes, et ille a raison. Cette personne crée donc un rapport le bug #43.

   Par la suite, soit vous soit elle s’en occupe :

   (a) Modifiez le fichier `main.c` pour avoir dans le main :

   ```
   1  char *name;
   2  if (argc < 2) {
   3    name = "tout le monde";
   4  }
   5  else {
   6    name = argv[1];
   7  }
   8  printf("Bonjour %s !\n", name);
   9  return 0;
   ```

   (b) Compilez le avec `gcc main.c`.

   (c) Testez le avec `./a.out Sam` et `./a.out`.

   → Validez la modification avec `git commit main.c -m "fix issue #43"`.

8. La feature est maintenant prête.

   On va donc basculer dans la branche master avec `git checkout master`.

   → Une fois que vous êtes dans la branche master, importez la nouvelle feature avec la commande `git merge` *votrenom*`-fr14`.

Exercice 6.

Historique amélioré et alias.

1. On a déjà vu comment visualiser l’historique de Git avec `git log`.

   Voici quelques astuces pour avoir un log plus sympa et informatif :

   — `git log --oneline`

   — `git log --oneline --decorate`

— `git log --oneline --decorate --graph`
— `git log --oneline --decorate --graph --all`
→ Qu'apporte chacun de ces arguments ?

2. Vous pouvez définir des alias pour les commandes que vous faites souvent.
   Par exemple :
   — `git config --global alias.s "status"`
   — `git config --global alias.ci "commit"`
   — `git config --global alias.lol "log --oneline --decorate --graph --all"`
   Cela vous permettra d'utiliser `git s` à la place de `git status` par exemple.