TP C avancé GIT
G. Subrenat

Licence Informatique L3 - 2020-2021
UFR SFA - Université de Poitiers

Git

Table des matières

1 Résumé et cadre de travail

Le but de ce TP est d'appréhender le logiciel *Git* pour une utilisation de base :
- principe du logiciel
- versionnement
- branches
- fusion de branches
- conflits
- divers

Dans ce document l'apprentissage de *Git* se fera par l'exemple.

Il existe des interfaces graphiques pour manipuler *Git*, par exemple *GitKraken* ou *Tortoise Git*. De même la plupart des IDE intègrent la gestion de *Git*.
C'est bien plus confortable et il faut les utiliser.
Mais dans ce TP nous utiliserons la ligne de commandes pour maîtriser toutes les étapes de *Git*.

Les commandes seront données pour un système Linux, mais elles seront quasiment identiques [1] pour les consoles de commandes des autres systèmes.

2 Principe de *Git*

2.1 Présentation

Git est un outil de versionnement. Autrement dit il garde une sauvegarde de toutes les versions d'un projet et permet de visualiser à volonté les anciennes versions.

Il permet de travailler à plusieurs sur un projet et aide à gérer les conflits si plusieurs développeurs modifient les mêmes fichiers.

Attention *Git* n'est pas un outil de gestion de projet, c'est uniquement un outil d'aide. En aucun il ne remplace un chef de projet et une gestion rigoureuse.

Git utilise des branches. Cela permet de développer des versions "parallèles" du code (pour développer et tester une fonctionnalité, pour explorer une solution, ...). Ensuite, si nécessaire, une nouvelle branche est fusionnée avec la branche "principale".

Git manipule des repositories (dépôts en français) distants ainsi qu'un repository local, ce qui sera étudié plus tard dans le document.

Git peut tout à fait être utilisé pour un projet géré par un seul développeur, et c'est même recommandé. Dans ce cas c'est la partie versionnement qui est utilisée (avec des branches si nécessaire).

Chaque version du projet concerne une seule fonctionnalité/action même si elle est minime. Il est fréquent qu'un développeur crée plus de dix versions par jour.
On ne fait pas une version pour faire une sauvegarde ; chaque version doit être un code correct (dans la mesure du possible).

2.2 Quiz : Principe de *Git*

Réponses page 26.

Question 1
plusieurs réponses possibles
Quel est le rôle de *Git* ?
- garder un historique de toutes les versions d'un projet
- travailler à plusieurs sur un projet
- faire de la gestion de projet

1. Voire complètement identiques

Question 2
une seule réponse possible
Il est intéressant d'utiliser *Git* pour un projet avec un seul développeur.
- vrai
- faux

Question 3
une seule réponse possible
À quelle fréquence crée-t-on des versions ?
- très rarement : un logiciel a très peu de versions dans son existence
- chaque fois qu'un fichier source est définitivement terminé
- une fois par jour pour faire une sauvegarde
- toutes les heures pour ne pas perdre plus d'une heure de travail en cas de panne
- très souvent, chaque fois qu'une fonctionnalité est implémentée et testée

Question 4
une seule réponse possible
Git est très peu utilisé.
- vrai
- faux

Question 5
une seule réponse possible
Il existe des logiciels comparables à *Git*.
- vrai
- faux

3 Initialisation de *Git*

3.1 Création d'un dépôt local

Hypothèses :
- Nous nous plaçons dans le cas où un seul utilisateur utilise *Git* et que la gestion est locale à la machine de l'utilisateur (le dépôt est local et il n'y a pas de dépôt distant).
- Nous partons d'un projet vide.

La première étape est de créer un répertoire et de se placer dedans [2].

```
$ mkdir PROJET_C
$ cd PROJET_C
$ pwd
/home/gilles/TP/PROJET_C
$ ls -lA
total 0
```

On peut alors créer le dépôt :

```
$ git init
Dépôt Git vide initialisé dans /home/gilles/TP/PROJET_C/.git/
$ ls -lA
drwxrwxr-x 7 gilles gilles 4096 oct. 24 20:48 .git
```

On note la création du répertoire *.git* qui contient toute la configuration du dépôt. Il contiendra aussi tout l'historique du projet.
Il ne faut pas modifier directement ce répertoire sinon on risque des destructions de données et/ou des dysfonctionnements graves. Toute action doit passer les commandes *git*.

2. histoire de commencer en douceur

Remarque : si l'on veut travailler sur un projet existant et hébergé sur une machine distante, il faut le rappatrier sur la machine locale avec la commande "*git clone*". Ensuite le fonctionnement que nous allons décrire sera le même.

Note : pour avoir une aide sur une commande *git*, la syntaxe est la suivante :

```
$ git help <cmd>
```

Par exemple [3] :

```
$ git help help
```

ou encore [4] :

```
$ git help init
```

Ça y est, le dépot est prêt, enfin presque.

3.2 Importation d'un dépôt distant

Hypothèses :

- Nous voulons intégrer un projet, géré par *Git*, qui est présent sur internet (repository distant).
- Nous partons donc d'un projet en cours de développement.

La première étape est d'importer le repository distant en local. Prenons comme exemple un projet de test de *gitlab* [5] :

```
$ git clone git@gitlab.com:gitlab-org/gitlab-test.git
Clonage dans 'gitlab-test'...
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Il faut avoir un compte paramétré sur *gitlab* pour rappatrier le projet [6].
Si c'est c'est le cas, voici le résultat :

```
$ git clone git@gitlab.com:gitlab-org/gitlab-test.git
Clonage dans 'gitlab-test'...
remote: Enumerating objects: 72, done.
remote: Counting objects: 100% (72/72), done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 1243 (delta 23), reused 27 (delta 5), pack-reused 1171
Réception d'objets: 100% (1243/1243), 10.72 MiB | 1.35 MiB/s, fait.
Résolution des deltas: 100% (563/563), fait.
Vérification de la connectivité... fait.
```

Note : lors d'une première connexion à *gitlab*, il peut y avoir un message d'avertissement :

```
The authenticity of host 'gitlab.com (172.65.251.78)' can't be established.
ECDSA key fingerprint is SHA256:LeK7q1yGdMDmcUfsNoHXPXv8Bw2b8efYRidmSRmGbCh.
Are you sure you want to continue connecting (yes/no)?
```

C'est normal. Il suffit de répondre par l'affirmative.

On peut alors descendre dans le répertoire fraîchement créé :

```
$ ls -l
drwxr-xr-x 12 gilles gilles 4096 2020-10-26 16:05 gitlab-test/
$ cd gitlab-test
$ pwd
```

3. l'équivalent du "*man man*"
4. les commandes *"git init --help"* et *"man git init"* sont équivalentes
5. https ://gitlab.com/gitlab-org/gitlab-test
6. Pour être plus exact, il suffit d'avoir une clé SSH déposée sur un compte actif de *gitlab*.

```
/home/gilles/TP/gitlab-test
$ ls -lA
drwxr-xr-x 8 gilles gilles 4096 2020-10-26 16:05 .git/
-rw-r--r-- 1 gilles gilles 100 2020-10-26 16:05 .gitattributes
-rw-r--r-- 1 gilles gilles 241 2020-10-26 16:05 .gitignore
-rw-r--r-- 1 gilles gilles 270 2020-10-26 16:05 .gitmodules
-rw-r--r-- 1 gilles gilles 22846 2020-10-26 16:05 CHANGELOG
...
drwxr-xr-x 2 gilles gilles 4096 2020-10-26 16:05 with space/
```

Pour envoyer des modifications sur le repository de *gitlab* il faut avoir des droits supplémentaires.

Que l'on ait créé un dépôt local ou importé un dépôt distant, le reste du document s'applique indifféremment.
La seule différence aura lieu lorsque des modifications seront apportées au projet : il y aura une commande supplémentaire pour transmettre les changements au dépôt distant et ainsi les partager avec les autres développeurs.

Reprenons les explications en se basant sur le dépôt local créé à la section précedente.

3.3 Configuration du dépôt

3.3.1 Identité

Lorsqu'une nouvelle version est créée[7], elle est associée au développeur qui l'a faite afin que les autres développeurs aient connaissance de l'auteur de chaque modification ; *Git* a donc besoin de connaître le propriétaire du dépôt local[8] (nom et adresse mail).

Pour le montrer, lançons la commande suivante (qui sert à créer une nouvelle version[9] et que nous détaillerons plus tard) :

```
$ git commit
*** Please tell me who you are.
Run
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
to set your account's default identity.
Omit --global to set the identity only in this repository.
```

Il faut lancer les deux commandes proposées. Utiliser l'option *--global* indique que cette configuration s'appliquera aux autres projets *Git* du compte utilisateur courant. Partons de l'hypothèse que nous voulons une configuration locale au projet en cours :

```
$ git config user.email "gilles@travail.fr"
$ git config user.name "Gilles"
```

Et maintenant :

```
$ git commit
Sur la branche master Validation initiale rien àvalider
```

Il n'y a plus d'erreur (et accessoirement *Git* nous indique qu'il n'est pas possible de créer une nouvelle version puisqu'il n'y a eu aucune modification).

3.3.2 Éditeur

Chaque fois que l'on crée une nouvelle version, il faut saisir un message explicatilf dans un éditeur de texte. L'éditeur choisi par défaut par *Git* pourrait ne pas convenir ; il est possible d'en choisir un autre.

Première solution avec la variable d'environnement *GIT_EDITOR* (à mettre dans le *.bashrc*) :

7. on parle de *commit*
8. Si le projet a été importé via un *git clone*, on parle malgré tout d'un dépôt local : le répertoire courant n'est qu'une copie du dépôt distant, et donc est bien local. Il y aura une commande spéciale (*git push*) pour transmettre les modifications locales vers le dépôt distant.
9. Notons que cela n'a pas de sens puisque nous n'avons encore rien rajouté dans le projet.

```
$ export GIT_EDITOR=vim.tiny
```

Seconde solution (que nous choisissons) avec le fichier de configuration de *Git* :

```
$ git config core.editor gedit
```

L'option -- *global* est possible.

La première solution est prioritaire sur la seconde. Le fichier de configuration local est prioritaire sur le fichier de configuration global.

3.3.3 Couleur

Pour activer la couleur dans les affichages de *Git*, le paramètre est *color.ui* qui peut prendre les valeurs *false*, *auto* (valeur par défaut), *true* et *always*. Par exemple :

```
$ git config color.ui true
```

3.3.4 Gestion de l'authentification

(attention le contenu de cette sous-section est incertain, à vérifier avec d'autres sources)

Lorsqu'on se connecte à des dépôts distants, ceux-ci demandent généralement une phase d'authentification qu'il faut potentiellement renseigner à chaque accès au dépôt. Il peut rapidement être pénible de devoir saisir fréquemment ses identifiants.

Git intègre un dispositif de gestion des identifiants avec le paramètre *credential.helper* (toujours avec la possibilité de l'option *--global*).

Pour Windows :

```
$ git config credential.helper wincred
```

Pour Linux :

```
$ git config credential.helper cache
```

Pour Mac :

```
$ git config credential.helper osxkeychain
```

Beaucoup de serveurs *Git* (dont *gitlab*) autorisent l'authentification par clés SSH, ce qui dispense des phases d'authentification explicites.

3.4 *.gitkeep* et *.gitignore*

3.4.1 *.gitkeep*

Git ne gère que les fichiers mais pas les répertoires, autrement dit il ignorera les répertoires vides.
La seule solution pour traquer un répertoire vide est ... qu'il ne soit pas vide.

L'habitude est que l'on mette à l'intérieur un fichier vide nommé *.gitkeep*[10].

Ce fichier peut être supprimé une fois que d'autres fichiers ont été ajoutés au répertoire.

3.4.2 *.gitignore*

Tous les fichiers n'ont pas vocation à être gérés par *Git*, i.e. n'ont pas être sauvegardés dans les dépôts. Par exemple :

10. N'importe quel autre nom de fichier aurait le même effet, ce n'est qu'une convention.

- les produits d'une compilation : fichiers objets, exécutables
- les fichiers de sauvegarde temporaires des éditeurs
- ...

Toutes les règles décrivant les fichiers ignorés doivent se trouver dans des fichiers nommés *.gitignore*. Il peut y avoir des fichiers *.gitignore* dans plusieurs répertoires ; la portée des règles d'un tel fichier est son répertoire ainsi que les sous-répertoires descendants.

Voici un exemple simple de fichier *.gitignore* :

```
# tous les fichiers objets
*.o

# les sauvegardes temporaires des éditeurs de texte
*~
*.bak
```

Voici un fichier *.gitignore* qui empêche tout le contenu du répertoire d'être géré par *Git* :

```
*
!.gitignore
```

On note la négation pour indiquer que le fichier *.gitignore* doit lui être géré par *Git*.
Nous n'irons, dans le cadre de ce document, pas plus loin dans les explications.

3.5 Pour aller plus loin

Les paramètres de configuration globale de *Git* se trouvent dans le fichier *$HOME/.gitconfig*
Les paramètres de configuration locale de *Git* se trouvent dans le fichier */.git/config*
Rappel : il faut éviter d'éditer ces fichiers à la main, mais plutôt passer par la commande *"git config"*.

Voici quelques commandes de configuration :

git config --list	:	liste tous les paramètres
git config --local --list	:	liste tous les paramètres locaux
git config --global --list	:	liste tous les paramètres globaux
git config --unset <nom-param>	:	supprime un paramètre local
git config --global --unset <nom-param>	:	supprime un paramètre global
git config <nom-param>	:	affiche la valeur du paramètre utilisé (cf. ci-dessous)
git config --local <nom-param>	:	affiche la valeur du paramètre local
git config --global <nom-param>	:	affiche la valeur du paramètre global

Commande : *git config <nom-param>* :

- si le paramètre n'est défini qu'en local ou (exclusif) qu'en global, la commande affiche la valeur,
- si le paramètre est défini à la fois en local et en global, la commande affiche la version locale.

3.6 Quiz : Initialisation de *Git*

Réponses page 26.

Question 1
une seule réponse possible
Un répertoire géré par *Git* doit toujours être lié à un dépôt distant géré par un sereur.

- vrai
- faux

Question 2
plusieurs réponses possibles
Quelle(s) commande(s) permet(tent) d'enregistrer son nom dans *Git* ?

- *git config user.name "mon nom"*
- *git init user.name "mon nom"*
- *git config --global user.name "mon nom"*

- *git init --global user.name "mon nom"*
- *git --global config user.name "mon nom"*
- *git --global init user.name "mon nom"*
- *export GIT_AUTHOR_NAME="mon nom"*

Question 3
une seule réponse possible
Je ne renseigne pas mon nom et mon email dans la configuration de *Git*.
- vrai : si je fais une bêtise, les autres développeurs ne pourront pas me retrouver
- faux : ainsi les autres développeurs sauront sur quelles parties je suis intervenu

Question 4
plusieurs réponses possibles
Je veux enregistrer un répertoire vide dans *Git*. Je mets dans le répertoire le fichier suivant :
- *.gitkeep* vide
- *.gitkeep* avec un espace dedans
- *.gitignore* avec une * dedans
- *.empty* vide
- *.vide* vide

Question 5
une seule réponse possible
Les fichiers *.gitignore* sont inutiles, autant sauvegarder un maximum d'informations dans les repositories *Git*.
- vrai
- faux

4 Gestion des versions

4.1 Préambule

Nous somme dans l'hypothèse où il n'y a qu'une seule branche (généralement appelée *master*) dans laquelle vont se succéder linéairement les différentes versions.

Nous nous plaçons dans la configuration suivante :
- Les données sont centralisées (cf. ci-dessous) dans un repository distant (par exemple hébergé par *gitlab*) qui sert de référence.
- Nous avons une copie dans le repository local (obtenu par *"git clone"*).
- Nous sommes le seul développeur actif. Autrement dit le repository distant n'est pas modifié pendant que nous travaillons en local. Ainsi nous ne gérons pas, dans un premier temps, les conflits [11] avec les autres développeurs.

Nous parlons de repository central, mais *Git* est un outil décentralisé. Ceci dit pour simplifier les explications nous supposons que :
- Il y a un repository central et distant qui sert de référence.
- Chaque développeur travaille dans son repository local et se synchronise uniquement avec le repository central.

Avertissement : pour l'instant nous avons uniquement un repository local (obtenu par *"git init"*) et nous ne pourrons pas faire les instructions manipulant le repository distant. Elles seront tout de même indiquées pour information.

La démarche pour créer une nouvelle version est la suivante :
- On ajoute et on modifie les fichiers nécessaires.
- On indique à *Git* quels fichiers font partie de la future nouvelle version (tous les fichiers créés ou modifiés ne sont pas obligatoirement à prendre en compte).

11. au sens *Git* du terme

- On crée la nouvelle version sur le repository local (*commit*).
- Si on utilise un repository distant, on lui envoie la nouvelle version (*push*).

Note : il est possible de faire plusieurs versions locales d'affilée pour les envoyer en une seule fois au repository distant (par exemple si on n'a pas accès à internet).

Plus en détails cela donne :

- Rappel : une version correspond à une seule fonctionnalité même si elle est minime.
- On vérifie qu'une fonctionnalité n'est pas en cours de développement [12] (cf. rappel ci-dessus) :

```
$ git status
```

- On part d'un état "propre", c'est à dire que l'on rappatrie la dernière version en provenance du repository central [13] [14] :

```
$ git pull
```

- On applique autant de fois que nécessaire les opérations suivantes :
  - création/modification de fichiers avec son IDE favori
  - tests et débuggage
- On vérifie les fichiers potentiellement concernés par la nouvelle version :

```
$ git status
```

- On indique à *Git* quels fichiers doivent réellement faire partie de la nouvelle version :

```
$ git add <liste fichiers>
```

  ou plus simplement si tous les fichiers sont concernés :

```
$ git add .
```

- On crée la nouvelle version sur le repository local :

```
$ git commit
```

- On envoie la nouvelle version sur le repository distant :

```
$ git push
```

4.2 Exemple pratique

Rappel il n'y a pas de repository distant, donc les affichages des commandes *Git* seront plus courts qu'avec un projet partagé.

4.2.1 Ajout du fichier *.gitignore*

Le dépôt est complètement vide, mais il faut prendre tout de suite les bonnes habitudes et on vérifie que l'état du dépôt est "propre".

```
$ git status
Sur la branche master

Validation initiale

rien à valider (créez/copiez des fichiers et utilisez "git add" pour les suivre)
```

Il faut théoriquement vérifier que l'on a la dernière version :

12. Même si un développeur doit savoir s'il a un développement en cours.
13. cf. explication ci-dessus sur la notion de central et décentralisé
14. Avec nos hypothèses, comme nous sommes le seul développeur, nous devons avoir de fait la dernière version en local.

```
$ git pull
fatal: Aucun dépôt distant spécifié. Veuillez spécifier une URL ou un nom
distant depuis lesquels les nouvelles révisions devraient être récupérées.
```

Et on obtient bien un message d'erreur adéquat.

On crée le fichier *.gitignore* :

```
$ gedit .gitignore
$ ls -lA
drwxrwxr-x 7 gilles gilles 4096 2020-10-27 17:34 .git/
-rw-rw-r-- 1 gilles gilles 31 2020-10-27 17:35 .gitignore
$ cat .gitignore
# tous les fichiers objets
*.o
```

Voyons l'état du repository local :

```
gilles@bureau:PROJET_C$ git status
Sur la branche master

Validation initiale

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        .gitignore

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez "gi
t add" pour les suivre)
gilles@bureau:PROJET_C$
```

Nous nous précipitons alors pour enregistrer notre première version :

```
gilles@bureau:PROJET_C$ git commit
Sur la branche master

Validation initiale

Fichiers non suivis:
        .gitignore

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents
gilles@bureau:PROJET_C$
```

En effet, il faut d'abord indiquer à *Git* l'ensemble des fichiers, parmi ceux qu'il propose, qu'il doit intégrer à la nouvelle version. En l'occurrence, comme on les veut tous (il n'y en a qu'un !), on indique la commande globale :

```
$ git add .
```

Mais la nouvelle version n'est toujours pas créée, voyons ce qu'il dit de l'état du repository :

```
gilles@bureau:PROJET_C$ git status
Sur la branche master

Validation initiale

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

        nouveau fichier : .gitignore

gilles@bureau:PROJET_C$
```

Le fichier *.gitignore* est maintenant indexé et nous pouvons enregistrer, en local, la nouvelle version :

```
$ git commit
[master (commit racine) b1bea7d] Création du fichier .gitignore : non prise en compte des .o
1 file changed, 2 insertions(+)
create mode 100644 .gitignore
```

Git a lancé l'éditeur choisi lors de la configuration (*gedit* dans le cas présent) pour écrire un message expliquant la nouvelle version. Il est **indispensable** de saisir un message le plus clair et complet possible,

et il ne faut pas hésiter à mettre plusieurs lignes.
Git interdit les messages vides, et des messages du style " " ou "modif" sont à proscrire.

Voici ce qu'a proposé l'éditeur de texte. Seule la première ligne a été saisie :

affichage éditeur de texte

```
Création du fichier .gitignore : non prise en compte des .o
# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
# Sur la branche master
#
# Validation initiale
#
# Modifications qui seront validées :
# nouveau fichier : .gitignore
#
```

Notons également le numéro associé à ce commit : b1bea7d.

Normalement le repository local est maintenant dans un état propre :

```
$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Il faut théoriquement envoyer notre version sur le repository distant pour que les autres développeurs y aient accès :

```
gilles@bureau:PROJET_C$ git push
fatal: Pas de destination pour pousser.
Spécifiez une URL depuis la ligne de commande ou configurez un dépôt distant en utilisant

    git remote add <nom> <url>

et poussez alors en utilisant le dépôt distant

    git push <nom>

gilles@bureau:PROJET_C$
```

Et on obtient bien un message d'erreur adéquat puisque nous n'avons pas de repository distant.

Pour la suite, tant que nous n'utiliserons pas de repository distant, nous passerons sous silence les phases de *pull* et de *push*.

4.2.2 Ajout du fichier *hello.c*

Il faut s'assurer que le repository local est propre et qu'on a bien rappatrié la dernière version si on utilise un repository distant (cf. section précédente).

Nous allons programmer un "hello world !" et le tester avant d'en faire une nouvelle version.
Rappelez-vous qu'on crée une nouvelle version avec un code correct et testé.

Voici ce que peut donner une session de travail :

```
gilles@bureau:PROJET_C$ gedit hello.c &
[1] 11893
gilles@bureau:PROJET_C$ gcc -Wall -Wextra -pedantic -std=c99 -o hello hello.c
hello.c: In function 'main':
hello.c:6:27: error: expected ';' before ':' token
     printf("Hello world!"):
                           ^
gilles@bureau:PROJET_C$ gcc -Wall -Wextra -pedantic -std=c99 -o hello hello.c
gilles@bureau:PROJET_C$ ./hello
Hello world!gilles@bureau:PROJET_C$ gcc -Wall -Wextra -pedantic -std=c99 -o hello hello.c
gilles@bureau:PROJET_C$ ./hello
Hello world!
gilles@bureau:PROJET_C$
```

Une fois la fonctionnalité terminée et testée [15], il faut créer une nouvelle version. Mais avant cela, il faut toujours vérifier l'état du repository local :

15. manifestement ce n'était pas inutile

```
gilles@bureau:PROJET_C$ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        hello
        hello.c

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez "gi
t add" pour les suivre)
gilles@bureau:PROJET_C$
```

Si on avait créé la nouvelle version sans vérification, l'exécutable y aurait été inclus. On s'aperçoit donc que le fichier *hello* ne doit pas intégrer la nouvelle version. Pour cela trois possibilités :

- on le supprime avant le *commit*
- on le rajoute au *.gitignore*
- on fait un *"git add"* sur mesure.

Nous choisissons la troisième solution, suivie d'un *commit* :

```
gilles@bureau:PROJET_C$ git add hello.c
gilles@bureau:PROJET_C$ git commit -m "Le sempiternel Hello World!"
[master d88c921] Le sempiternel Hello World!
 1 file changed, 9 insertions(+)
 create mode 100644 hello.c
gilles@bureau:PROJET_C$
```

Par la même occasion, nous voyons qu'il est possible d'intégrer directement un message court lors du *commit*, sans passer par l'éditeur de texte.

Notons également le numéro associé à ce commit : d88c921.

Et lorsqu'il y aura un dépôt distant, il faudra penser au *push*.

4.2.3 Suite du module *hello.c*

Nous allons étoffer notre projet avec un module *manip* (couple *manip.h* et *manip.c*) et un *Makefile*.

Tout d'abord le codage et les tests :

```
gilles@bureau:PROJET_C$ gedit hello.c manip.h manip.c Makefile &
[1] 12774
gilles@bureau:PROJET_C$ make
gcc -g -Wall -Wextra -pedantic -std=c99 -c -I. hello.c -o hello.o
gcc -g -Wall -Wextra -pedantic -std=c99 -c -I. manip.c -o manip.o
gcc -g -Wall -Wextra -pedantic -std=c99 -o hello hello.o manip.o -lm
gilles@bureau:PROJET_C$ ./hello
usage : ./hello <n>
   <n> : nombre de répétitions
message : nombre d'arguments incorrect
gilles@bureau:PROJET_C$ ./hello 3
Hello world!
Hello world!
Hello world!
gilles@bureau:PROJET_C$
```

Puis la vérification habituelle :

```
gilles@bureau:PROJET_C$ ls -lA
total 48
drwxrwxr-x 8 gilles gilles  4096 2020-10-28 16:09 .git/
-rw-rw-r-- 1 gilles gilles    31 2020-10-27 17:35 .gitignore
-rw-r--r-- 1 gilles gilles  2034 2020-10-28 15:52 Makefile
-rwxrwxr-x 1 gilles gilles 11840 2020-10-28 16:03 hello*
-rw-r--r-- 1 gilles gilles   537 2020-10-28 16:01 hello.c
-rw-rw-r-- 1 gilles gilles  6584 2020-10-28 16:03 hello.o
-rw-rw-r-- 1 gilles gilles   136 2020-10-28 15:58 manip.c
-rw-rw-r-- 1 gilles gilles    77 2020-10-28 15:55 manip.h
-rw-rw-r-- 1 gilles gilles  3440 2020-10-28 16:03 manip.o
gilles@bureau:PROJET_C$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         hello.c

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        Makefile
        hello
        manip.c
        manip.h

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
gilles@bureau:PROJET_C$
```

Quelques remarques :

- On voit bien la différences entre les fichiers modifiés et les fichiers nouvellement créés.
- Les *.o* n'apparaissent pas dans l'état du repository : c'est grâce au *.gitignore* créé plus tôt.

Nous allons supprimer tous les fichiers inutiles :

```
gilles@bureau:PROJET_C$ make distclean
rm -f hello.o manip.o
rm -f hello
gilles@bureau:PROJET_C$ ls -lA
total 24
drwxrwxr-x 8 gilles gilles 4096 2020-10-28 16:15 .git/
-rw-rw-r-- 1 gilles gilles   31 2020-10-27 17:35 .gitignore
-rw-r--r-- 1 gilles gilles 2034 2020-10-28 16:19 Makefile
-rw-r--r-- 1 gilles gilles  537 2020-10-28 16:01 hello.c
-rw-rw-r-- 1 gilles gilles  136 2020-10-28 15:58 manip.c
-rw-rw-r-- 1 gilles gilles   77 2020-10-28 15:55 manip.h
gilles@bureau:PROJET_C$
```

et créer la nouvelle version :

```
gilles@bureau:PROJET_C$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

        modifié :         hello.c

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

        Makefile
        manip.c
        manip.h

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
gilles@bureau:PROJET_C$ git add .
gilles@bureau:PROJET_C$ git commit -m "une bibliotèque pour les chaînes de catactères"
[master 59ea874] une bibliotèque pour les chaînes de catactères
 4 files changed, 160 insertions(+), 2 deletions(-)
 create mode 100644 Makefile
 create mode 100644 manip.c
 create mode 100644 manip.h
gilles@bureau:PROJET_C$
```

Notons le numéro associé à ce commit : 59ea874.

Nous nous apercevons, trop tard, qu'il y a une faute d'orthographe dans le message associé à cette nouvelle version.

Il est possible de corriger un message à condition :

- de le faire avant d'autres modifications,
- et surtout avant d'avoir envoyé la version sur le repository distant.

Ce type de commande est dangereux et peut conduire à des incohérences dans le repository ; à éviter si possible.

Voici la correction :

```
gilles@bureau:PROJET_C$ git commit --amend -m "une bibliotèque pour les chaînes de caractères"
[master ffcfa7e] une bibliotèque pour les chaînes de caractères
 Date: Wed Oct 28 16:23:00 2020 +0100
 4 files changed, 160 insertions(+), 2 deletions(-)
 create mode 100644 Makefile
 create mode 100644 manip.c
 create mode 100644 manip.h
gilles@bureau:PROJET_C$
```

On note que le numéro associé au commit a changé [16] : ffcfa7e.

4.3 Navigation dans les versions

Nous arrivons dans une partie particulièrement puissante de *Git* : consulter l'historique et naviguer dans les versions.

4.3.1 Afficher l'historique : *git log*

En premier lieu, rappelons que la commande suivante existe :

```
$ git help log
```

Le manuel fait près de 2000 lignes.

Voyons la commande de base :

```
gilles@bureau:PROJET_C$ git log
commit ffcfa7eaecac751613c0a4b4f14fed65404c2a87
Author: Gilles <gilles@travail.fr>
Date:   Wed Oct 28 16:23:00 2020 +0100

    une bibliotèque pour les chaînes de caractères

commit d88c921cb86e70174696d48072a222b3d492bffc
Author: Gilles <gilles@travail.fr>
Date:   Wed Oct 28 15:37:13 2020 +0100

    Le sempiternel Hello World!

commit b1bea7d83365b0862df980b7c60dae59366d38fc
Author: Gilles <gilles@travail.fr>
Date:   Tue Oct 27 19:08:36 2020 +0100

    Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Voici les informations affichées :

- Les versions s'affichent de la plus récente à la plus ancienne.
- Pour chaque version on a :
  - son numéro
  - son auteur
  - sa date
  - le message de description

L'option *--oneline* permet un affichage condensé :

16. Pour être exact le numéro du *commit* n'a pas changé mais un nouveau a été créé. Le *commit* 59ea874 existe toujours mais n'est plus rattaché à la branche *master*.

```
gilles@bureau:PROJET_C$ git log --oneline
ffcfa7e une bibliotèque pour les chaînes de caractères
d88c921 Le sempiternel Hello World!
b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Dans la suite du document, nous utiliserons fréquemment cette option afin de limiter la taille des captures d'écran.

L'option *--decorate* permet d'afficher sur l'historique :

- le nom des branches (une seule pour l'instant pour nous : *master*)
- sur quelle version on travaille : *HEAD* (lorsqu'on saura se déplacer dans l'historique)

```
gilles@bureau:PROJET_C$ git log --oneline --decorate
ffcfa7e (HEAD -> master) une bibliotèque pour les chaînes de caractères
d88c921 Le sempiternel Hello World!
b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Nous allons rapidement voir que cette notion de nom de branche et de *HEAD* est primordiale : ne pas parfaitement les comprendre peut conduire à des pertes de données (notion de *commit* orphelin).

L'option *--branches* avec sa compagne *--graph* permettent de visualiser :

- la totalité de la branche courante lorsqu'on se déplace dedans
- toutes (ou presques) les autres branches du projet.

Ce sera plus clair avec les exemples qui ne tarderont pas à venir. Dans l'état actuel, elles n'apportent pas de nouvelles informations.

4.3.2 La numérotation des versions

Chaque version est référencé par un identifiant unique en hexadécimal de 40 chiffres.
Actuellement nous avons les trois versions suivantes :

- ffcfa7eaecac751613c0a4b4f14fed65404c2a87
- d88c921cb86e70174696d48072a222b3d492bffc
- b1bea7d83365b0862df980b7c60dae59366d38fc

On appelle ces identifiants des SHA-1 (prononcer "cha ouane") car ils sont fabriqués par ... la fonction de hashage SHA-1.

C'est en précisant un tel numéro que l'on peut se déplacer dans l'historique.
On voit tout de suite la pénibilité de manipuler des identifiants aussi longs et peu intuitifs.
Aussi *Git* permet de manipuler ces identifiants en utilisant uniquement les premiers chiffres (au minimum 4, et généralement 7).

D'ailleurs lors des créations des versions (*commit*), les numéros n'étaient donnés qu'avec les 7 premiers chiffres.
La commande *"git log"* précise tous les chiffres.
La commande *"git log –oneline"* ne précise que 7 chiffres.

En ne précisant que les premiers chiffres, on pourrait avoir des conflits mais la probalité est faible (268 millions de possibilités avec 7 chiffres). En cas de conflit, il suffira de préciser plus de chiffres.

Notons qu'il est possible d'apposer des *tags* sur certaines versions afin de les manipuler avec des noms explicites.

4.3.3 Naviguer dans les versions

La commande est :

```
$ git checkout <sha-1>
```

Pour être précis, la commande *"git checkout"* permet aussi de naviguer dans les branches.

Par exemple nous allons revenir à la première version (celle où il n'y avait que le fichier *.gitignore*) qui a pour numéro *b1bea7d*.

Dans un premier temps, vérifions que nous sommes sur la version la plus récente :

```
gilles@bureau:PROJET_C$ git log --oneline --decorate
ffcfa7e (HEAD -> master) une bibliotèque pour les chaînes de caractères
d88c921 Le sempiternel Hello World!
b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$ ls -lA
total 24
drwxrwxr-x 8 gilles gilles 4096 2020-10-28 21:04 .git/
-rw-rw-r-- 1 gilles gilles   31 2020-10-27 17:35 .gitignore
-rw-rw-r-- 1 gilles gilles 2034 2020-10-28 21:04 Makefile
-rw-rw-r-- 1 gilles gilles  537 2020-10-28 21:04 hello.c
-rw-rw-r-- 1 gilles gilles  136 2020-10-28 21:04 manip.c
-rw-rw-r-- 1 gilles gilles   77 2020-10-28 21:04 manip.h
gilles@bureau:PROJET_C$
```

Analyse :

- C'est la référence (on parle aussi de pointeur [17]) *HEAD* qui indique sur quelle version nous sommes.
- Comment savoir que nous sommes sur la version la plus récente ? C'est la référence *master* qui nous l'indique. La référence *master* pointe toujours (par définition) sur la version la plus récente. Comme *HEAD* et *master* pointent sur la même version, alors la version courante est bien la plus récente.
- Le listing du répertoire montre bien les derniers fichiers.

Changeons maintenant de version :

```
gilles@bureau:PROJET_C$ git checkout b1bea7d
Note: checking out 'b1bea7d'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD est maintenant sur b1bea7d... Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Analyse :

- Pour l'instant nous laissons de côté la notion de *"detached HEAD"* ou de création d'une nouvelle branche. Disons juste que cela signifie qu'il ne faut pas faire des modifications et des *commit*, à moins de créer une branche.
- La dernière ligne nous rappelle le message associé à la version que nous venons de restaurer.

Le contenu du répertoire est maintenant :

```
gilles@bureau:PROJET_C$ ls -lA
total 8
drwxrwxr-x 8 gilles gilles 4096 2020-10-28 21:14 .git/
-rw-rw-r-- 1 gilles gilles   31 2020-10-27 17:35 .gitignore
gilles@bureau:PROJET_C$
```

Nous avons bien l'état du code source au moment de la première version.
Attention tout de même :

- Tous les fichiers qui ne sont pas intégrés à des versions resteront présents au fur et à mesure de la navigation
- Si des fichiers, déjà intégrés à la dernière version, sont en cours de modification, il ne sera pas possible de naviguer dans l'historique [18].

Que nous dit la commande *"git log"* ?

```
gilles@bureau:PROJET_C$ git log --oneline --decorate
b1bea7d (HEAD) Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Nous ne voyons que l'historique jusqu'à la version choisie, ce qui nous permet d'être dans les conditions de l'époque. Ceci dit il est souvent préférable de situer la version courante dans l'ensemble des versions ; une première utilité de l'option *--branches* (que nous associerons systématiquement à l'option *--graph*) apparaît alors.

17. Le terme de pointeur n'est pas un hasard comme nous le verrons dans une prochaine section.
18. cf. malgré tout, la commande : *git stash*

```
gilles@bureau:PROJET_C$ git log --decorate --oneline --branches --graph
* ffcfa7e (master) une bibliotèque pour les chaînes de caractères
* d88c921 Le sempiternel Hello World!
* b1bea7d (HEAD) Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Nous voyons que notre version courante (marquée par *HEAD*) est remontée de deux positions dans le temps par rapport à la version la plus récente de la branche courante [19] (marquée par *master*).

Après avoir regardé le code source de la version 1 de notre projet, il est temps de revenir à la version la plus récente pour continuer à développer.
Il y a alors deux commandes possibles :

```
$ git checkout ffcfa7e # NON
```

ou

```
$ git checkout master # OUI
```

La première est à proscrire [20]. Pour revenir en haut d'une branche, on passe toujours le nom de la branche à la commande *chekcout* et jamais le SHA-1 (nous expliquerons en détail ces subtilités dans une prochaine section) :

```
gilles@bureau:PROJET_C$ git checkout master
La position précédente de HEAD était sur b1bea7d... Création du fichier .gitignore : non prise en co
mptes des .o
Basculement sur la branche 'master'
gilles@bureau:PROJET_C$
```

Nous pouvons désormais continuer à développer le code dans la branche principale.

4.4 Quiz : Gestion des versions

Réponses page 27.

Question 1
une seule réponse possible
Git est un gestionnaire centralisé avec un dépôt principal et des dépôts secondaires.

- vrai
- faux

Question 2
plusieurs réponses possibles
La commande *"git status"* ?

- elle affiche le copyright du logiciel *Git*
- elle indique les changements effectués depuis la dernière version enregistrée
- elle indique les fichiers qui seront intégrés à la nouvelle version et ceux qui ne le seront pas
- elle indique si le projet est public ou privé et les coordonées du propriétaire

Question 3
plusieurs réponses possibles
Pourquoi la commande *"git commit"* crée une nouvelle version en local et ne la propage pas directement sur le repository distant ?

- Il n'existe pas obligatoirement un repository distant.
- Le système étant décentralisé, c'est au développeur de choisir sur quels dépôts distants il désire propager la nouvelle version.
- Le développeur peut ainsi faire des changements qui ne sont plus possibles une fois la version transmise.
- Le développeur n'a pas obligatoirement accès à internet.

Question 4
plusieurs réponses possibles

19. Rappelons que pour l'instant il n'y a qu'une seule branche, nommée par défaut *master*, dans notre projet.
20. sauf, comme toujours, si on sait ce que l'on fait

Pourquoi la commande *"git add"* est nécessaire et pourquoi tous les fichiers créés ou modifiés ne sont pas directement intégrés à une nouvelle version ?

- Cela réduit le trafic réseau.
- Cela réduit la place occupée sur disque.
- Cela permet au développeur d'avoir une double validation avant de créer la version.
- Des informations inutiles perturberaient les autres développeurs et génèreraient des versions inutiles.

Question 5
une seule réponse possible
Les SHA-1 pour identifier les versions sont trop risqués car il y a des risques de conflits (i.e. que deux versions aient le même identifiant).

- vrai
- faux

Question 6
une seule réponse possible
On peut identifier une version par autre chose qu'un SHA-1.

- vrai
- faux

Question 7
plusieurs réponses possibles
La commande *"git checkout"* permet de :

- naviguer dans l'historique des versions d'une branche
- changer la branche active
- supprimer physiquement une version devenue inutile
- créer une nouvelle branche

Question 8
une seule réponse possible
On peut modifier une version ancienne et faire des *commit* ?

- vrai
- faux

5 Fonctionnement détaillé des versions

5.1 Préambule

L'identification des versions avec des SHA-1 est assez claire, mais il reste des zones d'ombre au sujet des références que sont *HEAD* et *master* (et les noms des branches en général).

Les SHA-1 sont "physiquement" (i.e. indéfectiblement) liés à un *commit*[21] : il est impossible de changer le SHA-1 d'un *commit*.

Un nom de branche (*master* pour l'instant) n'est qu'un pointeur sur un *commit*. Et comme tout pointeur il est susceptible de se déplacer le long des *commit*.
Dans une utilisation normale[22], ce pointeur se déplace automatiquement et désigne toujours le *commit* le plus récent de la branche.

HEAD est un pointeur particulier :

- il se déplace à volonté
- il désigne la version de travail courante
- il peut pointer sur un SHA-1 (comme le fait un nom de branche)
- et plus perturbant (pour l'instant) il peut pointer sur un nom de branche (donc un pointeur sur un pointeur)

21. *commit* ≡ version et de fait *commit* ≡ version ≡ SHA-1
22. qu'il est fortement conseillé de suivre

Vraisemblablement ce n'est pas clair, et nous allons expliciter chaque point dans la suite.

5.2 Chaînage des *commit*

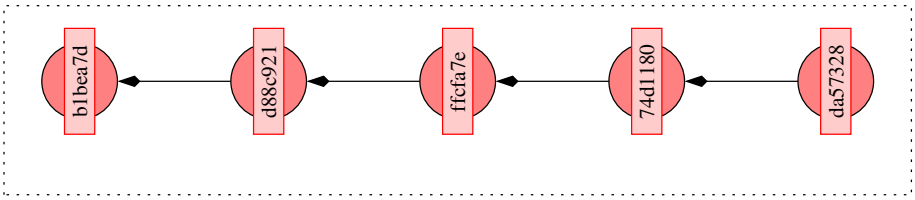
Sur une branche, les versions/*commit* sont reliés entre eux avec un chaînage arrière simple : la version la plus récente pointe sur l'avant-dernière qui elle-même pointe sur l'antépénultième et ainsi de suite jusqu'à la première version du projet.
Une autre façon est de dire que les versions sont gérées comme une pile.

Quand nous gérerons plusieurs branches, nous verrons que les versions forment un arbre chaîné "à l'envers", i.e. un fils pointe sur son père, et la racine de l'arbre est le premier *commit*.

Nous avons tout d'abord ajouté deux *commit* pour arriver à un total de 5 versions et rendre ainsi les schémas plus clairs :

```
gilles@bureau:PROJET_C$ git log --branches --graph --oneline --decorate
* da57328 (HEAD -> master) Ajout du fichier 0README
* 74d1180 Ajout d'une fonction encadrant un texte
* ffcfa7e une bibliotèque pour les chaînes de caractères
* d88c921 Le sempiternel Hello World!
* b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$ ./hello 3
Hello world!
Hello world!
Hello world!
+--------------+
| Hello world! |
+--------------+
gilles@bureau:PROJET_C$
```

Et voici le chaînage implémenté dans *Git* :



La première version est à gauche et la plus récence à droite.

5.3 Nom de branche

Si l'on pointe sur la version la plus récente, alors grâce au chaînage on peut revenir à n'importe quelle version de la branche.

En revanche, si on pointe sur une version intermédiaire, il n'est plus possible de revenir à la version la plus récente en se servant du chaînage.

Une solution serait de connaître le SHA-1 de la version la plus récente. Mais :

- c'est particulièrement pénible à se souvenir
- et il change à chaque nouvelle version.
- si on l'oublie, le projet est perdu [23].

Bref ce n'est pas la solution.

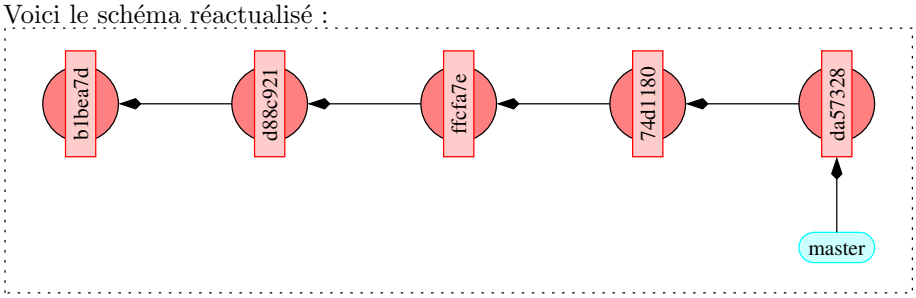
Aussi chaque branche a un pointeur dédié qui pointe en permanence sur le SHA-1 de la version la plus récente. Ce pointeur a pour nom le nom de la branche [24] : dans notre cas c'est *master*.

D'une part il est plus facile de se souvenir du nom des branches, d'autre part il existe une commande (*git branch*) qui donne la liste de toutes les branches.

Si on utilise correctement *Git* ce pointeur se déplace automatiquement chaque fois qu'une nouvelle version est créée.

23. Ce n'est pas tout à fait vrai, il est possible d'obtenir la liste de tous SHA-1.
24. En fait c'est le contraire : le nom de la branche est déterminé par le nom du pointeur.

Voici le schéma réactualisé :



Détruire un branche revient juste à supprimer le pointeur et en aucun cas les données.
Cependant si on détruit la branche (et c'est assez facile à faire), on ne peut plus accéder aux versions (à part des manipulations peu aisées, cf. commande *"git fsck"*). On parle alors de SHA-1 orphelins.

5.4 Pointeur *HEAD*

5.4.1 Principe de fonctionnement

Il reste à désigner la version courante de travail. Pour cela il y a un pointeur dédié nommé *HEAD*.

En règle général *HEAD* pointe sur la version la plus récente d'une branche, autrement dit au même endroit que le pointeur de branche (*master* pour l'instant dans notre cas) ; tout simplement parce que habituellement on travaille sur la dernière version d'un projet.

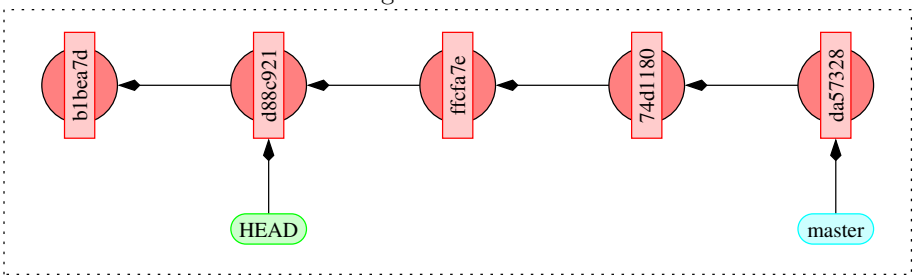
Mais nous avons vu, avec la commande *"git checkout"*, que *HEAD* pouvait pointer sur n'importe quel autre *commit* plus ancien.

Nous allons commencer par expliciter ce dernier cas, même si ce n'est pas le plus fréquent. Supposons que nous ayons tapé la commande :

```
gilles@bureau:PROJET_C$ git checkout d88c921
Note: checking out 'd88c921'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD est maintenant sur d88c921... Le sempiternel Hello World!
gilles@bureau:PROJET_C$
```

On note à nouveau un message anxiogène avec cette notion ésotérique de "detached HEAD".

Nous arrivons alors dans la configuration :

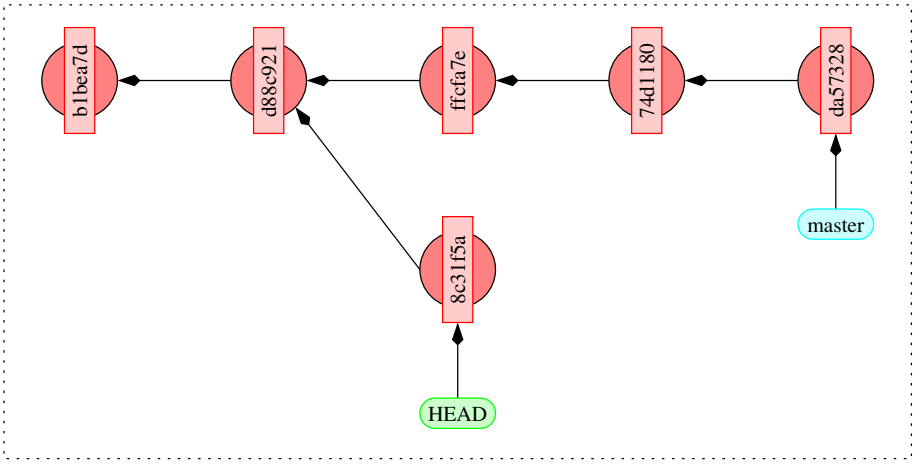


Rappelons que tous les nouveaux fichiers et toutes les modifications faits après cette version n'apparaissent plus dans les répertoires de travail.

5.4.2 *Commit* à partir d'une ancienne version : danger

A priori on ne peut pas faire de *commit* puisqu'il y a déjà une version après la version courante. Aussi généralement, lorsqu'on se positionne sur une version passée, on se contente de consulter.

Voyons cependant ce qu'il se passerait si on apportait une modification suivie d'un *commit* : il y aurait bien une nouvelle version qui n'écraserait pas la version *ffcfa7e* mais qui apparaîtrait à côté avec son propre SHA-1 (inventé pour l'occasion) :

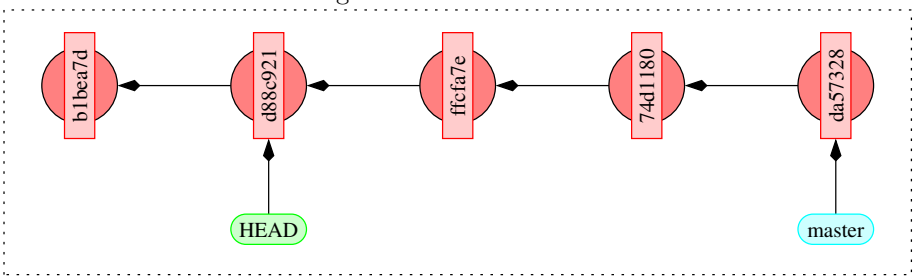


On voit que le pointeur *HEAD* s'est naturellement mis sur la nouvelle version. Mais le problème principal apparaît : lorsqu'on remettra le pointeur *HEAD* en bout de branche *master*, ce nouveau SHA-1 sera inaccessible (à moins de noter le SHA-1 sur un document extérieur ce qui n'est pas raisonnable).

L'expression "detached HEAD" est un peu plus claire : tous les *commit* créés seront détachés, autrement dit inaccessibles.

Lorsque nous aborderons les branches multiples, nous verrons qu'il est possible de rendre ces nouveaux *commit* accessibles.

Revenons à la situation d'origine :



5.4.3 Revenir à la dernière version : mauvaise solution

Il reste à revenir sur la dernière version de la branche *master* pour continuer à développer le projet. et il y a deux possibilités.

Commençons par la mauvaise :

```
gilles@bureau:PROJET_C$ git checkout da57328
La position précédente de HEAD était sur d88c921... Le sempiternel Hello World!
HEAD est maintenant sur da57328... Ajout du fichier 0README
gilles@bureau:PROJET_C$
```

Pourtant aucun message ne nous alerte. En réalité nous étions déjà dans le mode "detached HEAD" et nous le restons : *Git* ne nous met pas des messages alarmistes chaque fois.

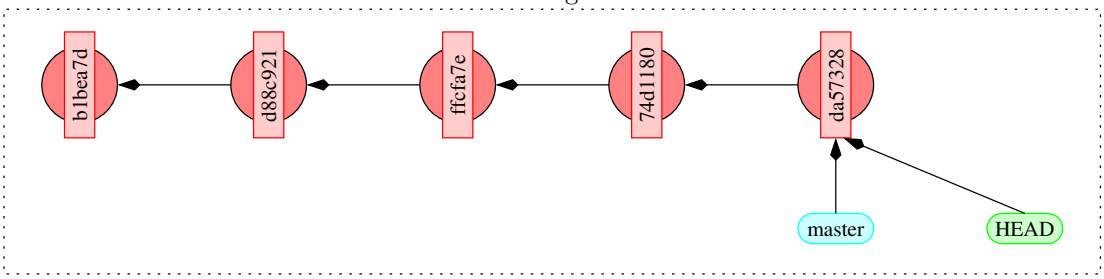
Regardons de près la commande *"git log"* :

```
gilles@bureau:PROJET_C$ git log --branches --graph --oneline --decorate
* da57328 (HEAD, master) Ajout du fichier 0README
* 74d1180 Ajout d'une fonction encadrant un texte
* ffcfa7e une bibliotèque pour les chaînes de caractères
* d88c921 Le sempiternel Hello World!
* b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

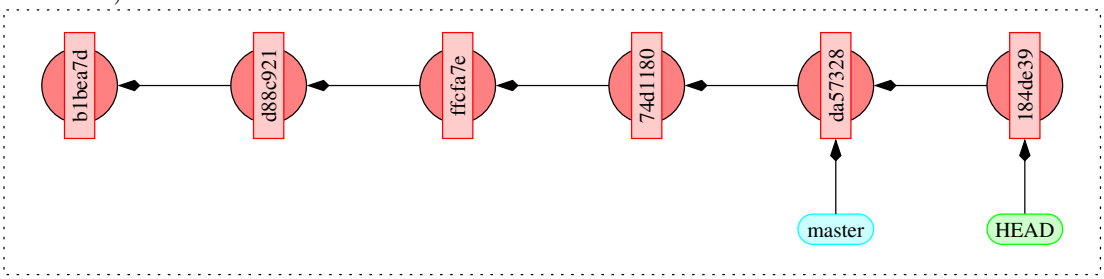
Et plus précisément la première ligne :

```
* da57328 (HEAD, master) Ajout du fichier 0README
```

HEAD et *master* sont séparés par une virgule ce qui signifie qu'ils sont indépendants bien que pointant sur le même *commit*. Nous sommes dans la configuration suivante :



Si nous faisons un nouveau *commit*, nous arrivons sur la configuration (avec un SHA-1 inventé pour l'occasion) :



Un premier problème est que *master* ne pointe pas sur la dernière version de la branche ce qui est incohérent avec un pointeur de branche.

Le deuxième problème (lié au premier) est que si on déplace *HEAD* sur une version passée, il n'y a plus moyen de revenir sur la dernière version (à moins de noter ce SHA-1 sur un document extérieur) : nous sommes exactement dans le même cas du *commit* fait précédemment à partir d'une version ancienne.

Bref pour revenir à la dernière version d'une branche, il ne faut pas utiliser le SHA-1 mais le pointeur de branche (c'est la deuxième et bonne solution).

5.4.4 Revenir à la dernière version : bonne solution

La bonne commande est la suivante :

```
gilles@bureau:PROJET_C$ git checkout master
La position précédente de HEAD était sur d88c921... Le sempiternel Hello World!
Basculement sur la branche 'master'
gilles@bureau:PROJET_C$
```

Notons que l'affichage nous dit qu'on bascule sur la branche.

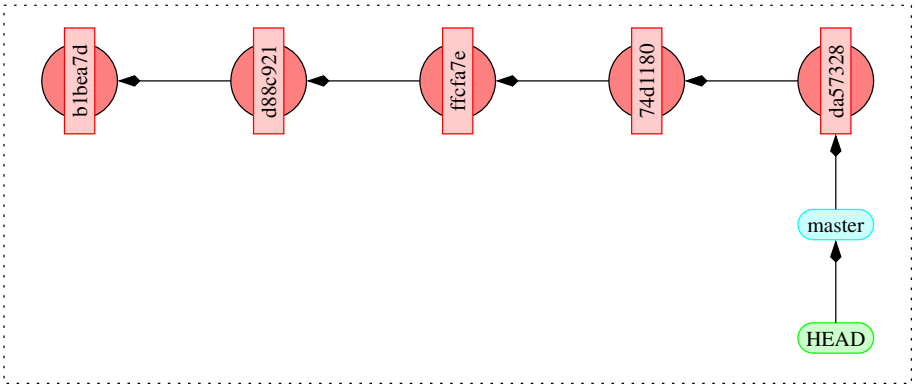
Regardons de près la commande *"git log"* :

```
gilles@bureau:PROJET_C$ git log --branches --graph --oneline --decorate
* da57328 (HEAD -> master) Ajout du fichier 0README
* 74d1180 Ajout d'une fonction encadrant un texte
* ffcfa7e une bibliotèque pour les chaînes de caractères
* d88c921 Le sempiternel Hello World!
* b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Et plus précisément la première ligne :

```
* da57328 (HEAD -> master) Ajout du fichier 0README
```

HEAD et *master* sont séparés par une flèche (et non plus une virgule) ce qui signifie que *HEAD* pointe sur *master* qui lui-même pointe sur la dernière version. Nous sommes dans la configuration suivante :



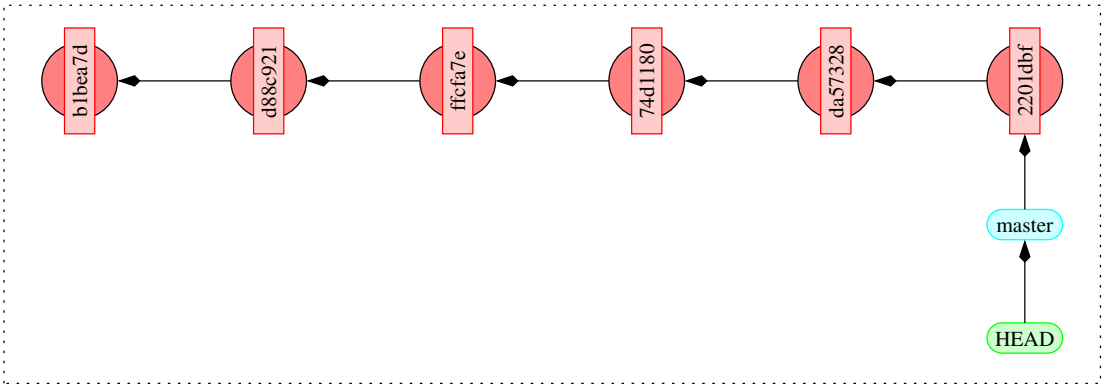
Faisons un nouveau *commit* :

```
gilles@bureau:PROJET_C$ git commit -m "Création d'un copyright"
[master 2201dbf] Création d'un copyright
 1 file changed, 1 insertion(+)
 create mode 100644 0COPYRIGHT
gilles@bureau:PROJET_C$
```

Et voyons le log :

```
gilles@bureau:PROJET_C$ git log --branches --graph --oneline --decorate
* 2201dbf (HEAD -> master) Création d'un copyright
* da57328 Ajout du fichier 0README
* 74d1180 Ajout d'une fonction encadrant un texte
* ffcfa7e une bibliotèque pour les chaînes de caractères
* d88c921 Le sempiternel Hello World!
* b1bea7d Création du fichier .gitignore : non prise en comptes des .o
gilles@bureau:PROJET_C$
```

Nous arrivons donc sur la configuration :



Et tout se passe bien car *HEAD* a transmis l'ordre au pointeur *master* de faire le *commit* ; *master* s'est déplacé sur la nouvelle version entraînant avec lui le pointeur *HEAD*.

En conclusion, pour éviter d'arriver à des configurations non voulues [25], on peut se fixer la règle suivante : "lorsqu'on fait un *commit*, le pointeur *HEAD* doit désigner un pointeur de branche (tel *master*) et jamais directement un SHA-1".

5.5 Quiz : Fonctionnement détaillé des versions

Réponses page 29.

Question 1

une seule réponse possible

On peut changer le SHA-1 d'une version (ou *commit*) afin par exemple d'avoir une numérotation plus logique.

- vrai
- faux

25. Sauf, comme d'habitude, si on sait ce que l'on fait.

Question 2
une seule réponse possible
Détruire une branche supprime définitivement toutes les versions qui y sont rattachées.

- vrai
- faux

Question 3
une seule réponse possible
Un pointeur de branche (comme *master*) désigne toujours le même *commit*, sinon on risquerait de perdre des données.

- vrai
- faux

Question 4
une seule réponse possible
Le pointeur *HEAD*, comme son nom l'indique, désigne toujours le premier *commit* d'une branche.

- vrai
- faux

Question 5
une seule réponse possible
Sur une branche le chaînage des *commit* est à l'envers : le premier *commit* de la liste chaînée est le dernier à avoir été créé.

- vrai
- faux

Question 6
une seule réponse possible
HEAD peut être un pointeur sur un pointeur, mais ça n'a pas de sens.

- vrai
- faux

Question 7
une seule réponse possible
Une mauvaise utilisation du pointeur *HEAD* peut conduire à un état incohérent du repository.

- vrai
- faux

6 Branches

7 Repositories distants

8 Réponses aux quiz

8.1 Réponses quiz : Principe de *Git*

cf. section 2.2, page 3

Question 1
plusieurs réponses possibles
Quel est le rôle de *Git* ?

- garder un historique de toutes les versions d'un projet
- travailler à plusieurs sur un projet
- ~~faire de la gestion de projet~~
  c'est au mieux une aide à la gestion de projet

Question 2
une seule réponse possible
Il est intéressant d'utiliser *Git* pour un projet avec un seul développeur.

- vrai
  pour la gestion de l'historique
- ~~faux~~

Question 3
une seule réponse possible
À quelle fréquence crée-t-on des versions ?

- ~~très rarement : un logiciel a très peu de versions dans son existence~~
  On ne parle pas des mêmes versions, et il est possible sous *Git* de marquer des versions particulières (au sens commercial du terme).
- ~~chaque fois qu'un fichier source est définitivement terminé~~
  Un fichier source n'est jamais définitivement terminé.
- ~~une fois par jour pour faire une sauvegarde~~
  Ce n'est pas un logiciel de sauvegarde (même s'il a cette conséquence) et il ne faut versionner que des états corrects.
- ~~toutes les heures pour ne pas perdre plus d'une heure de travail en cas de panne~~
  Cf. remarque précédente.
- très souvent, chaque fois qu'une fonctionnalité est implémentée et testée
  Une version correspond à une et une seule fonctionnalité.

Question 4
une seule réponse possible
Git est très peu utilisé.

- ~~vrai~~
- faux
  Au contraire il est très répandu dans le monde du logiciel et un développeur se doit de le connaître.

Question 5
une seule réponse possible
Il existe des logiciels comparables à *Git*.

- vrai
  Plus ou moins proches ; *Mercurial*, *Subversion (SVN)*, *CVS*, ...
- ~~faux~~

8.2 Réponses quiz : Initialisation de *Git*

cf. section 3.6, page 8

Question 1
une seule réponse possible
Un répertoire géré par *Git* doit toujours être lié à un dépôt distant géré par un sereur.

- ~~vrai~~
- faux
  Le répertoire de travail peut être le seul dépôt géré par *Git*, il est alors réservé à un seul utilisateur.

Question 2
plusieurs réponses possibles
Quelle(s) commande(s) permet(tent) d'enregistrer son nom dans *Git* ?

- *git config user.name "mon nom"*
  enregistrement pour le projet en cours
- ~~*git init user.name "mon nom"*~~
  non, *git init* sert à créer un projet vide
- *git config --global user.name "mon nom"*
  enregistrement pour tous les projets de l'utilisateur
- ~~*git init --global user.name "mon nom"*~~
  cf. deux réponses plus haut
- ~~*git --global config user.name "mon nom"*~~
  non, *"--global"* est une option de la sous-commande *config*, pas de la commande *git*.
- ~~*git --global init user.name "mon nom"*~~
  deux fois non
- *export GIT_AUTHOR_NAME="mon nom"*
  c'est une autre manière de faire qui est prioritaire sur les paramètres des fichiers de configuration

Question 3
une seule réponse possible
Je ne renseigne pas mon nom et mon email dans la configuration de *Git*.

- ~~vrai : si je fais une bêtise, les autres développeurs ne pourront pas me retrouver~~
  no comment
- faux : ainsi les autres développeurs sauront sur quelles parties je suis intervenu

Question 4
plusieurs réponses possibles
Je veux enregistrer un répertoire vide dans *Git*. Je mets dans le répertoire le fichier suivant :

- *.gitkeep* vide
  toutes les réponses fonctionnent, mais celle-ci est la plus communément utilisée.
- *.gitkeep* avec un espace dedans
- *.gitignore* avec une * dedans
- *.empty* vide
- *.vide* vide

Question 5
une seule réponse possible
Les fichiers *.gitignore* sont inutiles, autant sauvegarder un maximum d'informations dans les repositories *Git*.

- ~~vrai~~
- faux
  Des informations inutiles perturberaient les autres développeurs et génèreraient des versions inutiles. En général les fichiers générés automatiquement (compilation, cache) ne sont pas stockés dans les repositories, ainsi que les données personnelles (configuration d'une IDE) ou sensibles (mot de passe) du développeur.

8.3 Réponses quiz : Gestion des versions

cf. section 4.4, page 18

Question 1
une seule réponse possible
Git est un gestionnaire centralisé avec un dépôt principal et des dépôts secondaires.

- ~~vrai~~
- faux
  Git est bien décentralisé ce qui le rend à la fois complexe et puissant, mais nous n'étudierons pas cet aspect.

Question 2
plusieurs réponses possibles
La commande *"git status"* ?

- ~~elle affiche le copyright du logiciel *Git*~~
- elle indique les changements effectués depuis la dernière version enregistrée
- elle indique les fichiers qui seront intégrés à la nouvelle version et ceux qui ne le seront pas
- ~~elle indique si le projet est public ou privé et les coordonées du propriétaire~~

Question 3
plusieurs réponses possibles
Pourquoi la commande *"git commit"* crée une nouvelle version en local et ne la propage pas directement sur le repository distant ?

- Il n'existe pas obligatoirement un repository distant.
- Le système étant décentralisé, c'est au développeur de choisir sur quels dépôts distants il désire propager la nouvelle version.
- Le développeur peut ainsi faire des changements qui ne sont plus possibles une fois la version transmise.
  À utiliser avec circonspection ! Par exemple on supprime son dépôt local [26] et en refaisant un *"git clone"* afin d'oublier les dernières versions locales. Autre exemple, on utilise la commande *"git commit --amend"*. Il est bon de répéter que c'est à éviter au maximum.
- Le développeur n'a pas obligatoirement accès à internet.

Question 4
plusieurs réponses possibles
Pourquoi la commande *"git add"* est nécessaire et pourquoi tous les fichiers créés ou modifiés ne sont pas directement intégrés à une nouvelle version ?

- ~~Cela réduit le trafic réseau.~~
  Pour une nouvelle version, on choisit les fichiers selon des critères d'utilité et non de taille.
- ~~Cela réduit la place occupée sur disque.~~
  idem
- ~~Cela permet au développeur d'avoir une double validation avant de créer la version.~~
  Il se trouve que c'est une conséquence, mais ce n'est pas la raison.
- Des informations inutiles perturberaient les autres développeurs et génèreraient des versions inutiles.
  En général les fichiers générés automatiquement (compilation, cache) ne sont pas stockés dans les repositories, ainsi que les données personnelles (configuration d'une IDE) ou sensibles (mot de passe) du développeur.

Question 5
une seule réponse possible
Les SHA-1 pour identifier les versions sont trop risqués car il y a des risques de conflits (i.e. que deux versions aient le même identifiant).

- ~~vrai~~
  Le terme "trop risqué" n'est pas correct. Il y a théoriquement un risque mais il tellement faible qu'on le considère comme nul.
- faux

Question 6
une seule réponse possible
On peut identifier une version par autre chose qu'un SHA-1.

- vrai
  On peut apposer un (ou plusieurs) tag à une version particulière. Notons aussi l'identifiant particu-

26. À ne faire que si il existe un dépôt distant !

lier qu'est le nom d'une branche (*master* par exemple) et qui permet d'identifier la version la plus récente de la branche.

- ~~faux~~

Question 7

plusieurs réponses possibles

La commande *"git checkout"* permet de :

- naviguer dans l'historique des versions d'une branche
- changer la branche active
- ~~supprimer physiquement une version devenue inutile~~
  C'est impossible de supprimer physiquement une version : c'est l'essence de *Git* de garder une trace de tous les états et de pouvoir revenir dessus.
- créer une nouvelle branche
  Oui c'est possible avec l'option *"-b"* de *checkout* même si ce n'est pas la fonction première de cette commande.

Question 8

une seule réponse possible

On peut modifier une version ancienne et faire des *commit* ?

- vrai
  Mais à condition de créer une branche pour éviter les versions orphelines.
- faux
  Réponse incorrecte (cf. ci-dessus) mais acceptée au vu des connaissances acquises jusqu'à présent.

8.4 Réponses quiz : Fonctionnement détaillé des versions

cf. section 5.5, page 24

Question 1

une seule réponse possible

On peut changer le SHA-1 d'une version (ou *commit*) afin par exemple d'avoir une numérotation plus logique.

- ~~vrai~~
- faux
  On pourrait augmenter le risque d'avoir des collisions (dans le cas de projet à plusieurs milliers de *commit*). En outre les SHA-1 sont certainement utilisés en interne : chaînage des versions, position de *HEAD*, ...

Question 2

une seule réponse possible

Détruire une branche supprime définitivement toutes les versions qui y sont rattachées.

- ~~vrai~~
- faux
  Ce n'est pas possible de supprimer définitivement des données par définition d'un outil de versionnement. En revanche, en supprimant une branche, on peut rendre des *commit* orphelins et quasi-inaccessibles et de fait ils sont virtuellement supprimés.

Question 3

une seule réponse possible

Un pointeur de branche (comme *master*) désigne toujours le même *commit*, sinon on risquerait de perdre des données.

- ~~vrai~~
- faux
  Au contraire, un pointeur de branche bouge sans arrêt en pointant systématiquement sur le *commit* le plus récent.

Question 4

une seule réponse possible

Le pointeur *HEAD*, comme son nom l'indique, désigne toujours le premier *commit* d'une branche.

- ~~vrai~~
- faux
  Le premier *commit* est la racine de l'arbre et donc désigne toutes les branches du projet : avoir un pointeur dessus n'a pas beaucoup d'intérêt. *HEAD* pointe sur le *commit* de travail courant qui est potentiellement n'importe quel *commit*.

Question 5

une seule réponse possible

Sur une branche le chaînage des *commit* est à l'envers : le premier *commit* de la liste chaînée est le dernier à avoir été créé.

- vrai
  C'est bien le cas. À terme nous manipulerons des arbres et, avec un chaînage "à l'envers", chaque noeud n'a qu'un seul suivant. En outre ce sont les feuilles de l'arbre qui nous intéressent et donc avoir un accès direct dessus est intéressant ; en revanche il faut avoir autant de pointeurs qu'il y a de feuilles.
- ~~faux~~

Question 6

une seule réponse possible

HEAD peut être un pointeur sur un pointeur, mais ça n'a pas de sens.

- ~~vrai~~
- faux
  En l'occurrence c'est un point clé de *Git*, c'est ainsi que le pointeur de branche et *HEAD* peuvent se déplacer en ensemble.

Question 7

une seule réponse possible

Une mauvaise utilisation du pointeur *HEAD* peut conduire à un état incohérent du repository.

- vrai
  Et c'est même assez facile, soit en créant de nouveaux *commit* à partir de *commit* intermédiaires, soit en positionnant mal le pointeur *HEAD* lorsqu'on le remet sur la dernière version d'une branche.
- ~~faux~~