Dans ce TD, vous allez commencer à vous familiariser avec CVS : le *Concurrent Version System*. La commande principale pour cet utilitaire est `cvs`. Toutes les fonctionalités qu'il offre sont accessibles par des sous-commandes.

Commençons par définir une variable bash pour pointer vers le répertoire du TD :

```
export TD2=$HOME/1A-OMGL-VCS/TD2
```

puis il faut créer ce répertoire et nous y rendre :

```
mkdir -p $TD2
cd $TD2
```

Exercice 1. Création d'un repo que nous appelerons `$TD2/TD-CVS`. Ceci ce fait grace à la sous-commande `init`.

```
cvs -d $TD2/TD-CVS init
```

Une option qui se place avant la sous-commande est dite *globale*. L'option globale `-d $TD2/TD-CVS` permet d'indiquer à `cvs` où se trouve le repo qui nous intéresse.

Vous ne devez **jamais** travailler dans le dépôt. Le dépôt est essentiellement une base de données contenant l'historique. Si vous le modifiez à la main, vous massacrez la base de donnée et tout est foutu. Vous devez **toujours** travailler dans une *copie de travail* (notion reprise plus loin dans le TD).

Exercice 2. Commençons par créer une arborescence représentant un projet que nous aimerions gérer avec `cvs` :[1]

```
mkdir salut
mkdir salut/doc
mkdir salut/src
cat > salut/doc/README <<EOF
ce programme dit bonjour en plusieurs langues
EOF
cat > salut/src/salut.c <<EOF
int main()
{
}
EOF
```

Maintenant, *importons* ce projet dans `cvs` :

[1] si vous ne comprenez pas la notation `<<EOF` demandez au responsable de TD

```
cd salut
cvs -d $TD2/TD-CVS import -m "initialisation␣du␣projet" salut denys start
```

Vous pouvez vérifier que le projet a été importé dans `$TD2/TD-CVS` en allant regarder. Par contre, le répertoire courant est inchangé : il n'est pas encore sous contrôle de CVS (ceci se voit car il n'y a pas encore de sous-répertoire CVS). Commençons par renommer `salut` en `salut.orig` :

```
cd .. && mv salut{,.orig}
```

Exercice 3. Nous pouvons à présent obtenir une copie de travail du projet à partir de notre repo :[2]

```
cvs -d $TD2/TD-CVS checkout salut
```

Entrons dans le répertoire et examinons le :

```
cd salut
ls
```

Il y a maintenant un sous répertoire `CVS`. Celui-ci contient des fichiers administratifs et indique que l'arborescence ancrée ici est sous le contrôle de `cvs`. Dans le répertoire `CVS` vous trouverez les fichiers suivants : `Entries` qui indique que le répertoire principal du projet contient deux sous-répertoires : `doc` et `src` ; `Repository` qui indique que le *module* correspondant au répertoire principal dans le repo est `salut` ;[3] `Root` qui indique que le repo se trouve dans `$HOME/TD-CVS`.

Si vous regardez dans `salut/doc`, vous trouverez également un sous-répertoire `CVS` contenant : `Entries` qui indique qu'il y a un fichier `README` avec le numéro de version 1.1.1.1 ; `Repository` qui indique que le module correspondant dans le repo est `salut/doc` ; `Root` qui nous donne la même indication d'où se trouve le repo.

Exercice 4. Pour obtenir de l'aide :

```
cvs --help
```

Pour obtenir une liste des commandes (comme l'explique le résultat de la commande précédente) :

```
cvs --help-commands
```

Pour obtenir de l'aide sur une commande particulière, par exemple `status` :

```
cvs --help status              # methode 1
cvs -H status                  # methode 2
cvs status --help              # methode 3
```

Affichez le status de votre copie de travail :

```
cvs status
```

Notez que vous n'avez plus à spécifier `-d ~/TD-CVS` comme option globale car cette information est enregistrée dans `salut/CVS/Root`. Essayez les différentes options de cette commande.

[2] si vous ne comprenez pas cette manip bizarre (commande précédente et celle-ci), demandez au responsable de TD

[3] l'indication du module est utile car vous pouvez très bien faire un check-out d'un module `salut` dans un répertoire portant un autre nom, comme `bonjour`

Exercice 5. Modifiez `src/salut.c` pour qu'il retourne 0 comme il devrait. Voyez à présent ce que `status` vous dit :

```
cvs status
```

Maintenant, vous allez faire un *commit* de cette modification. Tout d'abord obtenez un peu d'aide sur la commande `commit` :

```
cvs --help commit
```

Puis invoquez cette commande en fournissant un message de log :

```
cvs commit -m "retourne␣0␣comme␣status"
```

Notez que si vous ne spécifiez pas en arguments les fichiers à enregistrer, alors `cvs` considère tous les fichiers modifiés. Voyons le `status` :

```
cvs status
```

Notez que tous les fichiers sont marqués "à jour" (Up-to-date) et que la version du fichier `src/salut.c` est à présent 1.2.

Exercice 6. Ajoutez un `#include <stdio.h>` en haut de `src/salut.c` et un `printf("bonjour\n");` dans le `main`. Puis ajoutez la ligne :

```
* francais
```

dans `doc/README`. Alors `status` nous dit les fichiers modifiés :

```
cvs status
```

Mais plus intéressant est de pouvoir visualiser exactement les modifications effectuées. La sous-commande `diff` permet de visualiser les différences entre l'original obtenu du repo et la version dans le répertoire de travail :

```
cvs diff
```

ou, pour obtenir la version "unifiée" :

```
cvs diff -u
```

Nous allons à présent faire un commit, mais nous allons utiliser un éditeur de texte pour rédiger le message de log :

```
cvs -e kwrite commit
```

On peut également utiliser les variables d'environnement `CVSEDITOR` ou `EDITOR` pour spécifier l'éditeur à utiliser par défaut.

Exercice 7. A présent ajoutez (au minimum) un `Makefile` dans le répertoire principal pour compiler votre projet.[4] Lorsque ce fichier est pret, ajoutez le au repo :

```
cvs add Makefile
```

Cette commande ne suffit pas : elle correspond à une modification de l'inventaire du projet, et, comme toute modification, il est nécessaire de faire un commit pour l'enregistrer dans le repo (cette fois-ci utilisons emacs comme éditeur) :

```
cvs -e emacs commit
```

Exercice 8. Vous pouvez consulter l'historique du projet :

```
cvs log
```

Vous pouvez également juste consulter l'historique d'un fichier particulier :

```
cvs log src/salut.c
```

Etudiez les options possibles pour cette commande :

```
cvs --help log
```

Il n'y a pas encore beaucoup d'historique pour les tester, mais vous pouvez quand même essayer.

Exercice 9. A partir de maintenant, vous allez travailler en groupes de 2 ou 3 (3 c'est mieux) sur le même projet. Choisissez l'un d'entre vous pour être le détenteur du repo partagé par votre groupe de travail. Supposons qu'il s'appelle `rikiki` : alors, si vous avez suivi mes instructions, son dépôt se trouve dans le répertoire `/home/rikiki/1A-OMGL-VCS/TD2/TD-CVS`. Définissons donc la variable bash `REPO` pour pointer vers ce dépôt :

```
export REPO=/home/rikiki/1A-OMGL-VCS/TD2/TD-CVS
```

Vous pouvez à présent obtenir une copie de travail du module `salut` de son repo :[5]

```
cvs -d $REPO co salut
```

ici `co` est une abbréviation permise de `checkout`. L'un d'entre vous (mais pas le détenteur du repo) doit alors apporter la modification suivante au `main` :[6] celui-ci doit (optionellement) prendre un argument (sur la ligne de commande). Cet argument est une chaîne de caractères indiquant le language désiré (pour dire bonjour) ; le défaut (si aucun argument n'est fourni) devrait être le français. Vous aurez besoin d'ajouter `#include <string.h>` et d'utiliser la fonction `strcmp` pour comparer l'argument avec les langages supportés par votre programme. Utilisez `man strcmp` pour découvrir comment vous servir de cette fonction (vous pouvez aussi demander de l'aide au responsable de TD).

Faites alors un commit de cette modif :

[4] c'est un projet pour de rire : donc rien n'a besoin de compiler. On fait juste comme si c'était un vrai projet, mais on ne s'embète pas à faire en sorte que les choses fonctionnent

[5] cette commande risque de ne pas fonctionner à cause des permissions par défaut : demandez au responsable de TD de vous aider à modifier ces permissions pour que tout fonctionne OK.

[6] je rapelle qu'on fait juste semblant : ce TD n'est pas un TD de programmation. Il n'est absolument pas nécessaire que le programme fonctionne ou même compile. On veut juste effectuer une série de modifications, comme si c'était un vrai projet.

```
cvs -e kwrite commit
```

Il est possible que ce commit ne fonctionne pas parce que vous n'avez pas les droits nécessaires pour le repo de votre collègue : dans ce cas, le détenteur du repo doit accorder des droits supplémentaires (soit au groupe, si vous êtes dans le même groupe unix, soit à tous les utilisateurs). Si ce problème se présente, demandez au responsable de TD : il résoudra ce problème une fois et l'écrira au tableau pour tout le monde. Ce problème se résoud grâce à l'utilisation de `chmod` pour changer les permissions dans le repo.

Exercice 10. Maintenant que `src/salut.c` a été modifié et commité[7] par l'un des développeurs du projet, il est nécessaire que les autre mettent à jour leur copie de travail : ceci ce fait grâce à la sous-commande `update`.

Mais il faut toujours procéder avec circonspection car une mise à jour pourrait causer de gros problèmes si elle est très incompatible avec vos propres modifs. Pour visualiser ce qu'un `update` ferait on peut l'invoquer avec l'option globale `-n`

```
cvs -n update
```

L'option globale `-n` est applicable à toutes les sous-commandes et en permet l'exécution "factice" pour voir ce qui se passerait, mais sans rien changer. L'exécution de la commande ci-dessus nous donne une liste de fichiers modifiés, chacun précédé d'un caractère :

`U FILE`	le fichier est mis à jour
`P FILE`	le fichier est mis à jour par un patch
`A FILE`	le fichier est ajouté
`R FILE`	le fichier est retiré
`M FILE`	le fichier est fusionné (merged) avec succès
`C FILE`	la fusion présente un conflit qu'il faudra résoudre
`? FILE`	le fichier est inconnu

Nous pouvons également visualiser les différences entre notre copie et la version la plus récente enregistrée dans le repo :

```
cvs diff -u -r HEAD
```

La révision `HEAD` est la révision la plus récente du *tronc* de dévelopement (the trunk) : le tronc, par opposition à une branche.

Bon, tout semble ok, alors nous effectuons l'update :

```
cvs up
```

ici `up` est une abbréviation permise de `update`.

La commande update est très importante : il vous faudra toujours faire un update avant de faire un commit car il est possible que pendant que vous faisiez vos modifs quelqu'un d'autre ait fait un commit. Si vous oubliez, cvs vous le rappelera en vous disant que vous n'êtes pas à jour.

[7] excusez le néologisme !

Exercice 11. Un autre dévelopeur maintenant ajoute la variable `langage`, de type chaîne de caractère, au `main` et l'initialise avec l'argument donné sur la ligne de commande s'il existe, avec `"francais"` sinon. Même chose que précédement : commit, puis update par tous.

Exercice 12. Indépendemment vous allez chacun ajouter le support pour un langage : vous testez la variable `langage` avec `strcmp` et si elle correspond au langage pour lequel vous implémentez le support alors vous faites un `printf` de la traduction correspondante. N'oubliez pas d'ajouter l'indication que ce langage est supporté dans le fichier `doc/README`.

Faites ensuite vos commits. Il est possible que lors d'un update vous rencontriez un conflit : ceci est signalé par update par le caractère `C` précédant le fichier dans lequel un conflit de fusion s'est produit. Un conflit sera marqué dans le fichier en question de la manière suivante :

```
<<<<<<< FICHIER
    MON-CODE
=======
    LE-CODE-DU-REPO
>>>>>>> LA-VERSION-DU-REPO
```

La résolution de ce conflit se fait en éditant le fichier pour ne garder que soit `MON-CODE`, soit `LE-CODE-DU-REPO`, soit une fusion des deux faite à la main. Dans tous les cas, il faut évidemment effacer les marqueurs "`<<<<<<< FICHIER`", "`=======`" et "`>>>>>>> LA-VERSION-DU-REPO`". On peut ensuite tenter à nouveau le commit.

Ajoutez d'autres langages (réels ou inventés pour l'exercice) pour bien prendre en main cette technique : modification + update + résolution de conflit + commit.