Outils de développement
TD n° 2
make et Makefile

Florent Devin

1 Préambule

Il a été fixé un certains nombres de règles, pour toutes les matières du département. Ces règles ne seront rappelées à chaque fois. Les règles qui ont été mises en place, ont plusieurs rôles :

- vous faciliter le travail ;
- vous sensibiliser à un travail professionnel ;
- vous prévaloir de problèmes.

Nous introduisons ici une structure qui vous rendra service tout au long de vos enseignements. Voici l'énoncé d'une partie des règles. Vous vous devez d'avoir :

- un répertoire par matière ;
- un sous-répertoire par activité ;
- un sous sous-répertoire par exercice ;
- ce dernier devra contenir obligatoirement un `Makefile` et les sous répertoires suivants :
  - `src` : qui contiendra les sources
  - `save` : qui contiendra une sauvegarde
  - `bin` : qui contiendra l'exécutable
  - `doc` : qui contiendra la documentation

Le `Makefile` doit pouvoir réaliser les cibles suivantes :

- `all` : pour compiler le programme. Le résultat de la compilation se trouvera dans le répertoire `bin`
- `clean` : pour effacer les fichiers générés. C'est à dire les fichiers du répertoire `bin`, qui ont été générés, ainsi que les fichiers créés par les éditeurs de texte (souvent ces fichiers portent l'extension : `*`$\sim$, `.bak`, `.old`, ou commencent et terminent par un `#`.
- `doc` : pour créer une documentation de votre programme. La documentation générée doit se trouver dans le répertoire `doc`. Si vous générez un *readme*, vous placerez ce fichier à la racine du projet/exercice.
- `save` : pour créer une sauvegarde de vos sources. Les fichiers sauvegardés se trouveront dans le répertoire `save`.

- `restore` : pour copier vos fichiers de sauvegarde contenu dans le répertoire `save`, vers le répertoire `src`.
- éventuellement `give` : pour créer une archive qui contiendra l'ensemble des fichiers nécessaires à la génération du projet, et de la documentation. Idéalement cette archive devra se décompresser dans un répertoire portant vos noms et prénoms sous la forme `nom_prenom`.

Nous allons donc dans ce TD, vous aider à rédiger un tel `Makefile`. Avec de la maîtrise, vous pourriez avoir envie de créer des `Makefile` qui automatiseront le déploiement de l'architecture telle qu'elle vous est demandée.

2 "Rappels"

Un projet ce compose souvent de plusieurs code sources. Comme vous le constaterez, on divise en général un projet en "brique". Chaque "brique", ou module, est composé de deux fichiers (un code source : fichier portant l'extension[1] `.c`, et un fichier autre portant l'extension `.h`). Pour construire le projet, il faut alors construire l'ensemble des modules, c'est à dire générer le fichier objet correspondant, puis construire le projet lui-même. C'est ce que l'on appelle la compilation séparée. Nous nous baserons sur cela pour le TD.

3 Exercices

Avant de commencer les exercices. Ne créez pas, pour l'instant, les répertoires cités ci-dessus, c'est à dire `src`, `doc`, `bin`, et `save`. Dans le répertoire du TD, créez les fichiers vides suivants : `score.c`, `score.h`, `gui.c`, `gui.h`, `ia.c`, `ia.h`. Créez un fichier `main.c` qui contiendra le programme minimum vu dans le cours de programmation C (Hello world). On supposera que le projet s'appelle `toto`

La commande `touch` permet de créer un fichier vide, et/ou de changer la date de dernière modification d'un fichier.

Dans un premier temps, vous n'utiliserez pas de variable.

Exercice 1 : Un `Makefile` "inutile"

Écrire un `Makefile` avec une seule règle capable de compiler le programme. Il ne doit y avoir compilation que si :

- l'exécutable n'existe pas
- l'exécutable est plus vieux que l'un des autres fichiers

Exercice 2 : Un pas vers la compilation séparée

Le `Makefile` précédent recompilait l'ensemble des sources à chaque changement. Cela est inutile, et peu s'avérer très coûteux. Pour remédier à cela, nous utilisons le principe

[1] Cela est valable pour les programmes C

de la compilation séparée. Pour chaque module, on génère le fichier objet associé. Pour générer le programme, il suffit juste d'avoir les fichiers objets à jour. Créez un `Makefile` qui respecte ces règles.

Exercice 3 : Un peu de nettoyage

Augmentez votre `Makefile` de la cible `clean`, qui effacera tous les fichiers "inutiles" (les fichiers objets et les backup des éditeurs de texte).

Il est vivement conseillé lorsque vous concevez une règle destructrice d'afficher la commande dans un premier temps, plutôt que de supprimer directement. Une fois que vous vous êtes assuré que la règle réalise bien ce que vous voulez, vous pouvez passer à l'action qui vous intéresse.

Pour supprimer un fichier, vous pouvez utiliser la commande `rm`. L'option `-i` demande confirmation avant de supprimer le fichier, l'option `-f` ne demande aucune confirmation.

L'appel `make clean` deux fois de suite ne doit pas provoquer d'erreur.

Exercice 4 : Utilisation de variables

Maintenant que vous avez un `Makefile` fonctionnel, créez les variables suivantes :

- `CC` : contient la commande qui lance le compilateur ⇔ `gcc -Wall`
- `RM` : contient la commande qui permet d'effacer
- `SRC` : contient l'ensemble des fichiers portant l'extension `.c`
- `HEAD` : contient l'ensemble des fichiers portant l'extension `.h`
- `OBJ` : contient la liste des fichiers objets qui doivent être générés
- `PROG` : le nom du programme final

L'intérêt d'utiliser des variables est de limiter les erreurs de recopie, mais surtout de concentrer les efforts de modification à un seul endroit. Utilisez ces variables ainsi que les méta variables vues en cours, pour rendre votre `Makefile` le plus général possible.

Exercice 5 : Utilisation des caractères joker

Si ce n'est pas déjà fait, utilisez *wildcard* pour créer les variables `SRC` et `HEAD`. Vous aurez peut être besoin d'utiliser la commande `filter-out`.

De même, utilisez les substitutions pour créer la variable `OBJ`.

Enfin, utilisez les suffixes pour permettre la compilation de chaque module.

Exercice 6 : Vérification de l'avancement

Vous commencez à avoir un `Makefile` utilisable, il vous permet déjà de compiler, et d'effacer les fichiers générer par la compilation. Par ailleurs, l'ajout d'un nouveau module est transparent. Pour vous en assurer, ajoutez les fichiers `save.c` et `save.h`, puis compilez. Normalement la prise en compte de ces fichiers doit être immédiate.

Créez un fichier `clean`. Que se passe-t-il lors de l'utilisation de la commande `make clean` ? Si `make` vous répond : `make : 'clean' is up to date.`, c'est que vous n'avez pas correctement écrit la cible `clean`. Corrigez ce problème.

Exercice 7 : Petit problème

Tout serait pour le mieux, si nous n'avions pas décider de vous obliger à respecter une structure, pour stocker vos données. Le problème ici vient du fait que vos fichiers sources se trouvent dans un répertoire, et les fichiers objets et le programme compilé dans un autre.

Normalement pour compiler les objets, vous avez utilisé une règle construite à partir de suffixes (`.sufx1.sufx2 :`). Ceci n'est pas adapté dans notre cas. Il vaut mieux utiliser les règles construites à base de motif. `GNU make` admet une autre écriture pour la règle cité précédemment. Elle peut s'écrire `%.sufx2 : %.sufx1`. Cette écriture est non seulement plus utile, comme nous allons le voir, mais aussi plus logique que la présente, en tout cas dans la manière d'écrire. En quoi la règle précédente est plus utile. Essayez de changer la règle `.c.o :` par `%.o : ./src/%.c`. Que constatez-vous ? Ceci peut être très utile pour résoudre notre problème.

Pour rendre plus clair votre `Makefile`, créez les variables suivantes :

- `srcdir` : répertoire qui contient les sources, et les entêtes du projet
- `bindir` : répertoire qui contient les objets, et exécutables du projet
- `docdir` : répertoire qui contient la documentation technique du projet
- `savedir` : répertoire qui contient les fichiers de sauvegarde du projet

Pour les variables `SRC` et `OBJ`, utilisez la définition suivante :

```
SRC = $(wildcard $(srcdir)*.c)
OBJ = $(subst $(srcdir), $(bindir),$(SRC:.c=.o))
```

Examinez et comprenez la règles qui permet de générer `OBJ`

Exercice 8 : Sauvegarde d'un projet

Créez une cible `save`, qui permettra de sauvegarder l'ensemble des fichiers sources dans le répertoire adéquat. Vous veillerez à ne copier que les fichiers utiles. Il peut être intéressant de créer une variable `CP` qui contient la commande permettant de faire la sauvegarde, à l'image de la variable `CC`.

Exercice 9 : Ce qu'il vous reste à faire

Vous venez de créer votre premier `Makefile`, pour qu'il soit utilisable, il faudra rajouter, les cibles `restore`, et `give`. Ne négligez pas les commentaires ! Ils vous permettront, par la suite, d'adapter ce fichier de configuration.

Exercice 10 : Pour aller plus loin

Si vous désirez aller plus loin, vous pouvez créer un `Makefile` dans le répertoire de la matière, qui s'occupera d'exécuter les `Makefile`s des sous répertoires.