Quelques éléments pour écrire des Makefile [1]

Make est un utilitaire permettant d'automatiser la compilation utile lorsque les sources sont organisés en plusieurs fichiers. Il permet de ne recompiler que les fichiers ayant été modifiés depuis la dernière compilation. Il se base sur un fichier décrivant les relations entre les fichiers et les actions à effectuer : le `Makefile`. Ce fichier doit s'appeler `Makefile` pour que make puisse le trouver automatiquement. Si ce fichier est présent dans le répertoire, pour compiler un programme il suffit de taper dans le terminal [2] :
`make`

Ce document a pour vocation de présenter rapidement ce qui pourra vous être utile dans le cadre de l'UE. Pour aller plus loin avec make vous pouvez lire :

— Manuel de GNU make http ://www.gnu.org/software/make/
— Managing Projects with GNU make. 3rd édition Andy Oratn, Steve Talbott, Robert Mecklenburg O'Reilly,

1 Makefile basique

Un Makefile est un fichier texte dans lequel on trouve principalement des déclarations de variables et des règles. Les déclarations de variables facilitent la lecture et la modification du Makefile, nous les verrons dans un second temps. Les règles décrivent le travail à effectuer par make. Elles doivent être écrites selon une syntaxe très stricte.

1.1 Syntaxe des règles

Une règle doit respecter la syntaxe suivante :
`<cibles>: <dépendances> <retour à la ligne>`
`<tabulation> <commande> <retour à la ligne>`
...
Exemple :

```
mon_prog: main.o fonction.o
     gcc -Wall -o mon_prog fonctions.o main.o
```

Les **cibles** sont les fichiers qui seront **générés par les actions (commandes)**. Il peut y avoir de 0 à plusieurs commandes à exécuter pour une cible donnée, et de 0 à plusieurs dépendances.
Note : on ne traitera ici que des Makefiles où la liste des cibles de chaque règle ne comporte qu'un seul fichier.

Les **dépendances** sont les **fichiers nécessaires à la génération des cibles**.

1. Ce document est fortement inspiré de celui disponible ici :`https://www.iut-info.univ-lille.fr/~beaufils/m3101/make.html`
2. Si on lui donne un autre nom, il faut lancer make avec l'option -f : `make -f MonMakefile`

Les **actions** sont les commandes à exécuter pour générer les cibles à partir des dépendances.

1.2 Exemple

Soit un programme dont la fonction `main` est dans un fichier nommé `main.c` et les autres fonctions sont dans un fichier nommé `fonctions.c` et leurs prototypes sont dans le fichier `fonctions.h`. Pour compiler ce programme et créer l'executable mon_prog il faut taper dans le terminal :

```
albert@monordi:~/$gcc -Wall -c fonctions.c
albert@monordi:~/$gcc -Wall -c main.c
albert@monordi:~/$gcc -Wall -o mon_prog fonctions.o main.o
```

Pour faire cela avec un makefile, il faut écrire dans le fichier `Makefile` :

```
mon_prog: main.o fonction.o
     gcc -Wall -o mon_prog fonctions.o main.o

fonction.o :  fonctions.c fonctions.h
     gcc -Wall -c fonction.c

main.o :  main.c fonctions.h
     gcc -Wall -c main.c
```

Nous avons défini dans ce fichier un règle permettant de créer l'executable mon_prog à partir des deux fichiers `.o` et deux **règles** permettant de générer deux fichiers `.o` (**cibles**) à partir de deux fichiers `.c` (**dépendances**).

Puis il faut taper dans le terminal :

```
albert@monordi:~/$make
```

1.3 Fonctionnement

Lorsqu'on tape `make` dans le terminal (sans argument), l'utilitaire prend la cible de la première règle comme cible principale [3].

Dans notre exemple, voici comment les choses se passent. Au départ, seuls les fichiers main.c, fonctions.c, fonctions.h et Makefile sont présents dans le répertoire avant le lancement de la commande make.

1. Make vérifie si l'executable mon_prog existe, ce n'est pas le le cas.
2. Comme les dépendances sont les fichiers main.o et fonction.o, il cherche dans le répertoire s'ils existent, mais ce n'est pas le cas.

3. On peut aussi préciser la cible principale explicitement : `make main.o` ne compilera que le fichier main.c

3. Il cherche dans le makefile une règle lui permettant de construire main.o et la trouve (règle 3). Il cherche les fichiers main.c et fonctions.h et les trouvent. Il exécute l'action `gcc -Wall -c main.c` correspondant à la règle 3. Le fichier main.o existe donc maintenant.
4. Make cherche une règle lui permettant de construire fonctions.o et la trouve (règle 2). Il cherche les fichiers fonctions.c et fonctions.h et les trouvent. Il exécute l'action `gcc -Wall -c fonction.c` correspondant à la règle 2. Le fichier fonctions.o existe donc maintenant.
5. Puisque les fichiers main.o et fonctions.o existent, make peut appliquer l'action `gcc -Wall -o mon_prog fonctions.o main.o` correspondant à la règle 1. Le fichier mon_prog existe donc maintenant : la cible principale a été générée, make s'arrête avec succès.

Imaginons que l'utilisateur relance immédiatement make.

1. Make vérifie si le fichier mon_prog existe et c'est le cas mais il va alors vérifier si certains fichiers ayant servi lors de la compilation de ce programme n'ont pas été modifiés depuis, *ie.* ont une date de modification plus récente que celle du programme.
2. Il cherche alors une règle permettant de construire main.o et trouve la règle 3 où les dépendances sont main.c et fonctions.h. Il cherche alors des règles permettant de construire main.c et fonctions.h et n'en trouve pas. Il considère donc que main.c et fonctions.h sont à jour. Il compare alors les dates de modification de main.c et fonctions.h avec main.o. Comme ce dernier est plus récent, il considère donc que main.o est à jour.
3. Il fait de même avec fonctions.o et fonctions.c et fonctions.h, d'après la règle 2.
4. Puisque main.o et fonctions.o sont à jour, make considère que prog l'est aussi.
5. Il s'arrête donc en affichant un message du type « `Target mon_prog is up to date.` »et aucune action n'est exécutée.

Imaginons maintenant que l'utilisateur modifie le fichier main.c et relance make. Make remonte jusqu'à main.c comme précédemment, mais cette fois-ci, main.o est plus ancien que main.c qui vient juste d'être modifié. Make exécute donc la commande `gcc -Wall -c main.c` pour mettre à jour main.o. Les fichiers fonctions.c et fonctions.h n'ayant pas été modifiés, make considère fonctions.o à jour. Mais comme main.o vient d'être régénéré, il est donc plus récent que mon_prog, qui n'est donc plus à jour. Donc make exécute la commande `gcc -Wall -o mon_prog fonctions.o main.o` pour mettre à jour mon_prog et s'arrête avec succès.

On voit donc que Make travaille sur le graphe des dépendances entre les fichiers, chaque noeud étant étiquetté avec sa date de dernière modification. On notera que ce graphe des dépendances ne doit pas contenir de cycle (eg : si A dépend de B qui dépend de C qui dépend de A).

1.4 Erreurs fréquentes

— Insérer des retours à la lignes superflus. Certains éditeurs de texte insèrent automatiquement des retours à la ligne lorsqu'une ligne de texte dépasse un certain nombre de caractères (autowrap). Pour couper une ligne trop longue, utiliser l'antislash (barre de fraction inversée \). Exemple :

```
cible: depl dep2 dep3 dep4 dep5 dep6 \
dep7 dep8 dep9
     commande
```

— Remplacer les tabulations par des espaces. Certains éditeurs de texte remplacent automatiquement les tabulations par un nombre d'espaces donné (soft tahs).

2 Makefile un peu moins basique

Vous pouvez vous arrêter ici si vous le souhaitez : vous savez maintenant écrire un Makefile basique. La suite vous apporte quelques éléments simples permettant d'éviter d'avoir à recopier de nombreuses fois des règles très similaires.

2.1 Variables

Make permet d'utiliser des variables dans un Makefile afin d'en faciliter la lecture et la mise à jour. Ainsi, on pourra écrire dans le Makefile (les lignes commençant par # sont des commentaires) :

```
#Les variables
#Le compilateur
CC = gcc
#Les options du compilateurs
CFLAGS = -Wall -g

#Les regles
mon_prog: main.o fonction.o
     $(CC)  $(CFLAGS) -o mon_prog fonctions.o main.o

fonction.o :  fonctions.c fonctions.h
     $(CC)   $(CFLAGS) -c fonction.c

main.o :  main.c fonctions.h
     $(CC)   $(CFLAGS)  -c main.c
```

Ceci permettra de changer facilement de compilateur C sans avoir à modifier chaque règle. Par convention, nous les avons écrit en majuscules mais il est possible d'utiliser tous les caractères ASCII sauf :, # et =.
Comme on le voit, un nom de variable doit être entouré de $( ) lorsqu'on veut utiliser sa valeur.

2.2 Cibles particulières

Il est parfois utile de définir des cibles qui ne sont pas des fichiers. Ainsi, pour supprimer les fichiers intermédiaires lors de la compilation précédente, on ajoute la règle :

```
clean :
    rm -f *.o mon_prog
```

Ainsi, taper `make clean` dans le terminal permet d'exécuter cette règle qui effacer tous les fichiers se terminant par `.o` et le fichier exécutable `mon_prog`.

Attention, si par malchance un fichier nommé `clean` existe dans votre répertoire, la cible `clean` sera considérée comme à jour et la commande ne sera alors pas exécutée ; make affichera le message «Target clean is up to date» et ne fera rien. Pour éviter ce comportement, il suffit de rajouter la directive `.PHONY: clean` au sommet du Makefile. Ainsi, make sait que clean n'est pas un fichier mais une cible particulière (phony target). Cette directive a de plus un effet de bord intéressant. Par défaut, make s'arrête dès qu'une erreur est détectée. Ainsi, s'il n'existe par exemple pas de fichier .o dans le répertoire courant, rm va renvoyer un code d'erreur et make s'arrêtera sans avoir effacé le fichier mon_prog. Avec la directive .PHONY, make exécute toutes les commandes de la règle correspondante et ignore les erreurs.

2.3 Règles implicites

Make est capable de générer certains fichiers même si on ne lui donne pas de règle explicitement. Cependant, l'utilisation de ces règles implicites induit des difficultés supplémentaires lors de la rédaction du Makefile. On désactivera ici cette fonction de make en utilisant la directive `.SUFFIXES`.

2.4 Raccourcis

Make permet d'utiliser des raccourcis pour éviter d'avoir à taper des longues listes de fichiers. On présente ici les trois raccourcis les plus utiles :

$@ représente la cible.

$^ représente la liste des dépendances.

$< représente la première dépendance (la plus à gauche dans la liste).

Exemple :

```
main.o :  main.c fonctions.h
     $(CC)  -Wall -c $<
```

Ces raccourcis sont surtout utiles lorsqu'on utilise des règles génériques.

2.5 Règles génériques

Dans l'exemple ci-dessus, on écrit deux règles très similaires pour générer main.o et fonctions.o :

```
fonction.o :  fonctions.c fonctions.h
     $(CC)  -Wall -c fonction.c

main.o :  main.c fonctions.h
     $(CC)  -Wall -c main.c
```

Pour éviter d'écrire ces deux règles, on peut définir une règle générique :

```
%.o : %.c %.h
     $(CC)  -c  $<
```

Cette règle peut être lue comme «chaque fichier .o dépend du fichier .c de même nom et peut être généré en utilisant la commande `$(CC) -c $<`. On remarque que l'on est ici obligé d'utiliser les raccourcis puisque les noms des fichiers sont variables.

2.6 Exemple complet d'un Makefile

```
#Ces cibles ne sont pas des vrais fichiers
.PHONY: clean all

#On désactive toutes les règles implicites
.SUFFIXES:

#Déclarations de variables
#Le compilateur
CC = gcc
#Les options du compilateurs
CFLAGS = -Wall -g
#Liste des programme à créer
PROGRAMS = mon_prog

#Premiere regle: liste des programme à compiler
#Règle sans action, seulement avec des dépendances
all: $(PROGRAMS)

#Règle pour compiler les programmes
mon_prog: main.o fonctions.o
    $(CC) $(CFLAGS) -o $@ $^

#Règle générique  de compilation des .o à partir des .c
%.o: %.c %.h
    $(CC)  $(CFLAGS) -c  $<

#Effacer les .o et les executables
#Pour executer cette regle il faut taper dans le terminal "make clean"
clean:
    rm -f *.o $(PROGRAMS)
```