Ecriture d'un `Makefile`

ENSIMAG 1A

Année scolaire 2008–2009

Un `Makefile` est un fichier, utilisé par le programme `make`, regroupant une série de commandes permettant d'exécuter un ensemble d'actions, typiquement la compilation d'un projet. Un `Makefile` peut être écrit à la main, ou généré automatiquement par un utilitaire (exemples : `automake`, `gmake`, ...). Il est constitué d'une ou de plusieurs règles de la forme :

```
cible: dépendances
       commandes
```

Lors du parcours du fichier, le programme `make` évalue d'abord la première règle rencontrée, ou celle dont le nom est spécifié en argument. L'évaluation d'une règle se fait récursivement :

1. les dépendances sont analysées : si une dépendance est la cible d'une autre règle, cette règle est à son tour évaluée ;
2. lorsque l'ensemble des dépendances a été analysé, et si la cible est plus ancienne que les dépendances, les commandes correspondant à la règle sont exécutées.

1 Exemple

Ce qui suit présente la création d'un `Makefile` pour un exemple de projet. Supposons pour commencer que ce projet regroupe trois fichiers `exemple.h`, `exemple.c` et `main.c`.

1.1 `Makefile` de base

Un fichier `Makefile` de base de ce projet pourrait s'écrire :

```
mon_executable: exemple.o main.o
	gcc -o mon_executable exemple.o main.o

exemple.o: exemple.c
	gcc -o exemple.o -c exemple.c -Wall -O

main.o: main.c exemple.h
	gcc -o main.o -c main.c -Wall -O
```

En effet pour créer l'exécutable `mon_executable`, nous avons besoin des fichiers objets `exemple.o` et `main.o`. `make` va d'abord rencontrer `exemple.o`, et va donc évaluer la règle dont ce fichier est la cible. La seule dépendance de cette règle étant `exemple.c`, qui n'est pas cible d'une autre règle, `make` va exécuter la commande :

```
gcc -o exemple.o -c exemple.c -Wall -O
```

De la même façon, `make` va ensuite analyser la règle dont `main.o` est la cible et exécuter la commande :

```
gcc -o main.o -c main.c -Wall -O
```

Finalement, toutes les dépendances de la règle initiale ayant été analysées, `make` va exécuter la commande :

```
gcc -o monexecutable exemple.o main.o
```

1.2 Un premier raffinement

Pour améliorer ce `Makefile`, on peut rajouter quelques cibles standards :
- `all` : à placer généralement au début du fichier ; les dépendances associées correspondent à l'ensemble des exécutables à produire ;
- `clean` : normalement pas de dépendance ; la commande associée supprime tous les fichiers intermédiaires (notamment les fichiers objets) ;
- `mrproper` : la commande correspondante supprime tout ce qui peut être regénéré, ce qui permet une reconstruction complète du projet lors de l'appel suivant à `make`.

Notre `Makefile` devient donc ici :

```
all: mon_executable

mon_executable: exemple.o main.o
		gcc -o mon_executable exemple.o main.o

exemple.o: exemple.c
		gcc -o exemple.o -c exemple.c -Wall -O

main.o: main.c exemple.h
		gcc -o main.o -c main.c -Wall -O

clean:
		rm -f *.o core

mrproper: clean
		rm -f mon_executable
```

1.3 Introduction de variables

Il est possible de définir des variables dans un `Makefile`. Elles se déclarent sous la forme `NOM=valeur` et sont appelées sous la forme `$(NOM)`, à peu près comme dans un shellscript. Parmi quelques variables standards pour un `Makefile` de projet C ou C++, on trouve :
– `CC` qui désigne le compilateur utilisé ;
– `CFLAGS` qui regroupe les options de compilation ;
– `LDFLAGS` qui regroupe les options d'édition de liens ;
– `EXEC` ou `TARGET` qui regroupe les exécutables.

Pour notre projet exemple, cela donne :

```
CC=gcc
CFLAGS=-Wall -O
LDFLAGS=
EXEC=mon_executable

all: $(EXEC)

mon_executable: exemple.o main.o
                $(CC) -o mon_executable exemple.o main.o $(LDFLAGS)

exemple.o: exemple.c
                $(CC) -o exemple.o -c exemple.c $(CFLAGS)

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

clean:
                rm -f *.o core

mrproper: clean
                rm -f $(EXEC)
```

Il existe aussi, et c'est encore plus intéressant car très puissant, des variables *internes au* `Makefile`, utilisables dans les commandes ; notamment :
– `$@` : nom de la cible ;
– `$<` : nom de la première dépendance ;
– `$^` : liste des dépendances ;
– `$?` : liste des dépendances plus récentes que la cible ;
– `$*` : nom d'un fichier sans son suffixe.

On peut donc encore compacter notre `Makefile` :

```
CC=gcc
CFLAGS=-Wall -O
LDFLAGS=
```

```
EXEC=mon_executable

all: $(EXEC)

mon_executable: exemple.o main.o
	$(CC) -o $@ $^ $(LDFLAGS)

exemple.o: exemple.c
	$(CC) -o $@ -c $< $(CFLAGS)

main.o: main.c exemple.h
	$(CC) -o $@ -c $< $(CFLAGS)

clean:
	rm -f *.o core

mrproper: clean
	rm -f $(EXEC)
```

1.4 Règles d'inférences

On voit néanmoins qu'il y a encore moyen d'améliorer ce `Makefile` : en effet les règles dont les deux fichiers objets sont les cibles se ressemblent fortement. Or justement, on peut créer des règles génériques dans un `Makefile` : il suffit d'utiliser le symbole `%` à la fois pour la cible et pour la dépendance. Il est également possible de préciser séparement d'autres dépendances pour les cas particuliers, par exemple pour ne pas oublier la dépendance au fichier `exemple.h` pour la règle dont `main.o` est la cible. Ceci nous donne :

```
CC=gcc
CFLAGS=-Wall -O
LDFLAGS=
EXEC=mon_executable

all: $(EXEC)

mon_executable: exemple.o main.o
	$(CC) -o $@ $^ $(LDFLAGS)

main.o: exemple.h

%.o: %.c
	$(CC) -o $@ -c $< $(CFLAGS)

clean:
	rm -f *.o core

mrproper: clean
```

```
	rm -f $(EXEC)
```

1.5 Liste des fichiers objets et liste des fichiers sources

On peut encore simplifier ce `Makefile`. En effet, on constate que la génération de l'exécutable dépend de tous les fichiers objets. Ceci peut être long à écrire dans le cas d'un gros projet ! Pour simplifier l'écriture, sachant que tous les fichiers objets correspondent aux fichiers sources en remplaçant l'extension `.c` par l'extension `.o`, on peut utiliser les variables `SRC` et `OBJ` et définir `OBJ=$(SRC :.c=.o)`. `SRC` sera définie par la liste des fichiers sources ... ce qui n'a fait que déplacer le problème ! Là encore, une syntaxe particulière permet de simplifier le `Makefile` : `SRC=$(wildcard *.c)`. `wildcard` permet l'utilisation de la variable `$*` en dehors d'une commande.

Le `Makefile` du projet exemple devient finalement :

```
CC=gcc
CFLAGS=-Wall -O
LDFLAGS=
EXEC=mon_executable
SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)

all: $(EXEC)

mon_executable: $(OBJ)
	$(CC) -o $@ $^ $(LDFLAGS)

main.o: exemple.h

%.o: %.c
	$(CC) -o $@ -c $< $(CFLAGS)

clean:
	rm -f *.o core

mrproper: clean
	rm -f $(EXEC)
```

Il existe encore bien d'autres possibilités pour simplifier un `Makefile`. Il est notamment possible de créer des `Makefile` pour des sous-répertoires correspondant à des sous-parties du projet, et d'avoir un `Makefile` "maître" très simple qui appelle ces "sous-`Makefile`", avec la variable `MAKE`. Il existe également des outils de génération automatique.