Introduction aux makefiles.*

Structures de données - Travaux dirigés sur machines

Introduction aux makefiles

Table des matières

1 Présentation de Makefile

Un *Makefile* est un fichier texte, sans extension, de nom `Makefile` ou `makefile`, permettant de décrire la manière dont doivent être compilés et liés un ensemble de fichiers sources afin de générer un programme exécutable.
Un *Makefile* est constitué de plusieurs règles de la forme :

```
cible: dependances
    commandes
```

Chaque commande est précédée d'une **tabulation** (Attention, copier/coller depuis ce fichier pdf peut remplacer la tabulation par des espaces et rendre le fichier Makefile incorrect)

Lors de l'utilisation d'un tel fichier, via la commande `make`, la première règle rencontrée, ou bien la règle dont le nom est spécifié, est évaluée. L'évaluation d'une règle se fait en plusieurs étapes :

— Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du *Makefile*, cette règle est à son tour évaluée.

*Memento extrait de Introduction à Makefile

— Lorsque l'ensemble des dépendances a été analysé et si la cible ne correspond pas à un fichier existant, ou si une dépendance est plus récente que la cible, les différentes commandes pour générer la cible sont exécutées.

1.1 Makefile minimal

Le *Makefile* minimal d'un projet contenant les fichiers sources `hello.h`, `hello.c` et `main.c` est donc le fichier contenant :

```
hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c -std=c99 -Werror -Wextra -Wall -ansi -pedantic

main.o: main.c hello.h
    gcc -o main.o -c main.c -std=c99 -Werror -Wextra -Wall -ansi -pedantic
```

Regardons de plus près sur cet exemple comment fonctionne un *Makefile* :

— Nous cherchons à créer le fichier exécutable `hello`, la première dépendance est la cible d'une des règles de notre `Makefile`, nous évaluons donc cette règle. Comme aucune dépendance de `hello.o` n'est une règle, aucune autre règle n'est à évaluer pour compléter celle-ci.

— Deux cas se présentent ici : soit le fichier `hello.c` est plus récent que le fichier `hello.o`, la commande est alors exécutée et `hello.o` est construit, soit `hello.o` est plus récent que `hello.c` est la commande n'est pas exécutée. L'évaluation de la règle `hello.o` est terminée.

— Les autres dépendances de `hello` sont examinées de la même manière puis, si nécessaire, la commande de la règle `hello` est exécutée et `hello` est construit.

1.2 Makefile enrichi

Plusieurs cas ne sont pas gérés dans l'exemple précédent :

— Un tel *Makefile* ne permet pas de générer plusieurs exécutables distincts.

— Les fichiers intermédiaires restent sur le disque dur même lors de la mise en production.

— Il n'est pas possible de forcer la régénération intégrale du projet

Ces différents cas conduisent à l'écriture de règles complémentaires :

— `all` : généralement la première du fichier, elle regroupe dans ces dépendances l'ensemble des exécutables à produire.

— `clean` : elle permet de supprimer tout les fichiers intermédiaires.

— `mrproper` : elle supprime tout ce qui peut être régénéré et permet une reconstruction complète du projet.

En ajoutant ces règles complémentaires, notre *Makefile* devient donc :

```
all: hello

hello: hello.o main.o
    gcc -o hello hello.o main.o

hello.o: hello.c
    gcc -o hello.o -c hello.c -std=c99 -Werror -Wextra -Wall -ansi -pedantic
```

```
main.o: main.c hello.h
    gcc -o main.o -c main.c -std=c99 -Werror -Wextra -Wall -ansi -pedantic

clean:
    rm -rf *.o

mrproper: clean
    rm -rf hello
```

2 Définition de variables

2.1 variables personnalisées

Il est possible de définir des variables dans un *Makefile*, ce qui rend les évolutions bien plus simples et plus rapides.
Il est souvent nécessaire, dans le cadre du développement multi-plateforme par exemple, de pouvoir compiler le code avec différents compilateurs.
En utilisant une variable pour désigner le compilateur à utiliser, plus besoin de changer l'ensemble des règles si le compilateur change, seule la variable correspondante est à modifier.
Une variable se déclare sous la forme `NOM=VALEUR` et s'évalue via l'expression `$(NOM)`.
Nous allons donc définir quatre variables dans notre *Makefile* :

- Une désignant le compilateur utilisée nommée `CC` (une telle variable est typiquement nommé `CC` pour un compilateur `C`, `CXX` pour un compilateur `C++`).
- `CFLAGS` regroupant les options de compilation (Généralement cette variable est nommées `CFLAGS` pour une compilation en `C`, `CXXFLAGS` pour le `C++`).
- `LDFLAGS` regroupant les options de l'édition de liens.
- `EXEC` contenant le nom des exécutables à générer.

Nous obtenons ainsi :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

$(EXEC): hello.o main.o
    $(CC) -o hello hello.o main.o $(LDFLAGS)

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

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

clean:
    rm -rf *.o

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

2.2 Variables internes

Dans la syntaxe des *Makefile*, il existes plusieurs variables internes pouvant être utilisées. Parmis celle-ci, les variables internes principalement utilisées sont :

- `$@` : Le nom de la cible
- `$<` : Le nom de la première dépendance
- `$^` : La liste des dépendances
- `$?` : La liste des dépendances plus récentes que la cible
- `$*` : Le nom du fichier sans suffixe

en utilisant ces variables, on peut réécrire notre *Makefile* de la façon suivante :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

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

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

clean:
    rm -rf *.o

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

3 Les règles d'inférence

Dans le dernier exemple ci-dessus, nous pouvons remarquer une forte ressemblance entre les règles de compilation.
Pour éviter cette redondance et rendre un *Makefile* plus facilement maintenable, un *Makefile* permet également de créer des règles génériques - ou règles d'inférence - (par exemple construire un `.o` à partir d'un `.c`) qui se verront appelées par défaut.
Une telle règle se présente sous la forme suivante :

```
%.o: %.c
    commandes
```

Il est alors possible de réunifier les différentes règles de compilation via des règles par défaut pour simplifier notre *Makefile* :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
```

```
all: $(EXEC)

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

$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

clean:
    rm -rf *.o

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

En utilisant les règles génériques comme ci-dessus, il manque alors des dépendances et le fichier `main.o` ne sera plus reconstruit si `hello.h` est modifié.
Il est alors nécessaire de compléter les règles d'inférences par des règles spécifiques permettant de décrire ces dépendances.

Le fichier *Makefile* de notre exemple est donc maintenant :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

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

$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)

# Specific file dependencies
main.o: hello.h
```

4 Les cibles prédéfinies

Dans l'exemple précédent, `clean` est la cible d'une règle ne présentant aucune dépendance.
Supposons que `clean` soit également le nom d'un fichier présent dans le répertoire courant, il serait alors forcément plus récent que ses dépendances et la règle ne serait alors jamais exécutée.
Pour pallier ce problème, il existe une cible particulière nommée `.PHONY` dont les dépendances seront systématiquement reconstruites.

En utilisant cette cible prédéfinie, nous obtenons donc :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

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

$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

.PHONY: clean mrproper

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)

# Specific file dependencies
main.o: hello.h
```

5 Génération de la liste des fichiers objets.

Plutôt que d'énumérer la liste des fichiers objets dans les dépendances de la règle de construction de notre exécutable, il est possible de la générer automatiquement à partir de la liste des fichiers sources.
Pour cela nous rajoutons deux variables au *Makefile* :

- `SRC` qui contient la liste des fichiers sources du projet.
- `OBJ` pour la liste des fichiers objets.

Seule la variable `SRC` devra être renseignée en indiquant la liste des sources. La variable `OBJ` est alors remplie à partir de la variable `SRC` de la façon suivante :

```
OBJ= $(SRC:.c=.o)
```

Pour chaque fichier `.c` contenu dans `SRC` nous ajoutons à `OBJ` un fichier de même nom mais portant l'extension `.o`.

Nous obtenons alors le Makefile :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= hello.c main.c
OBJ= $(SRC:.c=.o)

all: $(EXEC)

%.o: %.c
```

```
    $(CC) -o $@ -c $< $(CFLAGS)

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

.PHONY: clean mrproper

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)

# Specific file dependencies
main.o: hello.h
```

Il est à noter que l'on peut aussi remplir automatiquement la variable `SRC` à partir de la liste des fichiers sources (`*.c`) contenus dans le répertoire courant.
Cela ne peut pas se faire directement en utilisant `*.c` mais doit se faire en utilisant une comande spacifique du *Makefile* permettant d'utiliser les caractères génériques `*` et `?` : `wildcard`.

```
SRC= $(wildcard *.c)
```

6 Commandes silencieuses

Lorsque le nombre de fichier à compiler augmente, il peut alors devenir difficile de retrouver des messages d'erreur parmi la liste des fichiers compilés.
Pour simplifier l'affichage produit par `make` lors de la construction d'un projet, un *Makefile* permettent de désactiver l'écho des lignes de commande en rajoutant le caractère `@` devant la ligne de commande, notre *Makefile* devient alors :

```
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
SRC= hello.c main.c
OBJ= $(SRC:.c=.o)

all: $(EXEC)

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

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

.PHONY: clean mrproper

clean:
    @rm -rf *.o

mrproper: clean
    @rm -rf $(EXEC)
```

```
# Specific file dependencies
main.o: hello.h
```

Il est important de noter que, lorsque cela est nécessaire, il est possible de réactiver l'echo des comandes du Makefile sans modifier le Makefile, en exécutant la compilation par la ligne de commande

```
make VERBOSE=1
```

7 Makefile conditionnels

Les *Makefile* offrent une multitude de possibilité pour configurer la compilation d'un projet. Nous pouvons, par exemple, gagner en souplesse de compilation par l'utilisation de directives, assez proches des directives de compilation du C, qui permettent d'exécuter conditionnellement une partie du *Makefile* en fonction de l'existence d'une variable, de sa valeur, etc.
Supposons, par exemple, que nous souhaitions compiler notre projet soit en mode mise au point (*debug*), soit en mode production (*release*) sans avoir à modifier plusieurs lignes du *Makefile* pour passer d'un mode à l'autre.
Il suffit alors de créer une variable `DEBUG` et tester sa valeur pour changer de mode de compilation, i.e. ajouter l'option `-g` ou les options `-O3 -DNDEBUG` à la variables `CFLAGS`. L'exemple ci-dessous propose donc un *Makefile* très générique, permettant de compiler tous les fichiers sources d'un répertoire pour générer un exécutable donné. Notez que le symbole `#` permet de définir des commentaires dans le *Makefile*.

```
# Specific part of the Makefile
EXEC=hello

# Begin generic part of the Makefile
CC=gcc
CFLAGS=-std=c99 -Werror -Wextra -Wall -ansi -pedantic
LDFLAGS=
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

ifeq ($(DEBUG),yes)
    CFLAGS += -g
    LDFLAGS =
else
    CFLAGS += -O3 -DNDEBUG
    LDFLAGS =
endif

all: $(EXEC)

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

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

.PHONY: clean mrproper
```

```
clean:
    @rm -rf *.o

mrproper: clean
    @rm -rf $(EXEC)
# End generic part of the makefile

# List of target dependencies
main.o: hello.h
```

A partir d'un tel *Makefile*, il est alors possible en ne modifiant que très peu de choses de réutiliser le *Makefile* pour d'autres projet.

Les étudiants sont encouragés à approfondir leurs connaissances du *Makefile* à partir de la documentation de l'implantation la plus utilisée : GNU `make`.
Il est à noter aussi que les fichiers de description de projets spécifiques aux différents environnements de développement intégrés (EDI tels que CLion, qtcreator, CodeBlocks, visual studio, xcode, eclipse, ...) sont la plupart du temps construits sur le même principe de règles que les *Makefile*.
Toutefois, devant la complexité d'écriture d'un Makefile pour un projet complexe contenant quelques miliers de fichiers sources répartis entre plusieurs répertoires et générant plusieurs bibliothèques et éxécutables, certains outils de description de projet de plus haut niveau d'abstraction ont été développés.
Le programme cmake, permettant de générer des *Makefile* pour tout système d'exploitation et pour les EDI majeurs définit ainsi une syntaxe plus simple pour décrire la composition et les dépendances d'un projet. Cmake est de plus en plus utilisé dans l'industrie et dans le domaine du logiciel libre. Bien que dépassant les objectifs de l'UE Structures de données, nous encourageons les étudiants à étudier comment construire leurs projets en utilisant cmake.