Epitech Documentation

Makefiles

Les beaux Makefiles c'est cool

PART 1 - A SIMPLE MAKEFILE

STEP 1 - UN EXEMPLE DE PROJET : HELLO WORLD

```c
// hello.c

#include <unistd.h>

void say_hello(void)
{
    write(1, "Hello world !n", 14);
}
```

```c
// hello.h

#ifndef H_HELLO
    #define H_HELLO

    void say_hello(void);

#endif
```

```c
// main.c

#include "hello.h"

int main(void)
{
    say_hello();
    return 0;
}
```

Step 2 - La syntaxe des Makefiles

Les Makefiles sont des fichiers composés de plusieurs règles suivant cette forme :

```
cible:  dependances
    commandes
```

Lors de l'exécution de la commande make, la première règle rentrée, ou la règle spécifiée est évaluée. L'évaluation d'une règle se fait en suivant ces étapes :

- Les dépendances sont analysées, si une dépendance est la cible d'une autre règle du Makefile, cette règle est à son tour évaluée.
- Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier parmi les dépendances est plus récent que le fichier cible, les différentes commandes sont exécutées.

Attention, les commandes doivent impérativement être précédées d'une tabulation.

Step 3 - Un Makefile minimal

On peut donc écrire une première version minimale d'un Makefile pour notre mini projet :

```
# Makefile

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

hello.o: hello.c
    gcc -o hello.o -c hello.c -Wall -Wextra -Werror

main.o: main.c hello.h
    gcc -o main.o -c main.c -Wall -Wextra -Werror
```

Découpons l'exécution lors de l'exécution de la commande `make` :

- La première règle évaluée est la première rencontrée, soit "hello".
- La première dépendance de cette règle fait référence à une autre règle du Makefile, elle va donc être évaluée.
- La dépendance de la règle "hello.o" n'est pas une autre règle, mais un fichier, deux cas de figures se présentent alors :
    - Soit le fichier hello.c est plus récent que le fichier cible hello.o, alors la commande sera exécutée.
    - Dans le cas contraire, la commande ne sera pas exécutée.
- Les mêmes étapes sont appliquées pour la deuxième dépendance de la règle "hello" et elle-même. Ainsi, la commande ne sera exécutée que si un des fichiers hello.o ou main.o est plus récent que l'exécutable hello.

Part 2 - Un Makfile plus évolué

Dans la version écrite précédemment, plusieurs choses posent problème :

- Il ne permet pas la création de plusieurs exécutables.
- Les fichiers temporaires restent présents (.o).
- Les exécutables ne peuvent pas être supprimés efficacement.
- Il n'est pas possible de forcer la génération intégrale du projet.

Ces problèmes trouvent leur solution dans l'ajout de plusieurs règles :

- La règle `all`, généralement la première du fichier, elle a pour dépendance, l'ensemble des exécutables à générer.
- La règle `clean`, elle permet de supprimer les fichiers temporaires.
- La règle `fclean`, elle permet de supprimer l'intégralité des fichiers générés, incluant les fichiers temporaires et les exécutables.
- La règle `re`, elle permet de regénérer l'ensemble du projet.

Voilà à quoi ressemble notre Makefile maintenant :

```
# Makefile

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 -Wall -Wextra -Werror

main.o: main.c hello.h
    gcc -o main.o -c main.c -Wall -Wextra -Werror

clean:
    rm -rf *.o

fclean: clean
    rm -rf hello

re: fclean all
```

STEP 2 - UTILISATION DES VARIABLES

Il est en effet possible de définir des variables dans un Makefile, cela les rend plus agréables à lire, mais aussi à faire évoluer.
Les variables se déclarent sous la forme NOM = VALEUR et s'utilisent via la syntaxe $(NOM).
Nous allons donc pouvoir rajouter des variables à notre Makefile :

```
# Makefile

CC = gcc
CFLAGS = -Wall -Wextra -Werror
LDFLAGS =
EXEC = hello

all: $(EXEC)

hello: 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

fclean: clean
    rm -rf hello

re: fclean all
```

On a ainsi défini plusieurs variables :

- CC compilateur utilisé pour la compilation du C.
- CFLAGS flags utilisés lors de la compilation du C.
- LDFLAGS flags utilisés lors de l'invocation du linker.
- EXEC contient le nom des exécutables à générer.

Attention, les variables CC, CFLAGS et LDFLAGS sont des variables implicites utilisées par défaut par make. Pour plus de détails, voir Gnu.org

STEP 3 - LES VARIABLES INTERNES

Comme en shell script, il existe des variables internes au Makefile :

Variable	Correspondence
$@	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

On peut donc simplifier notre Makefile grâce à ces variables, ce qui nous donne :

```
# Makefile

CC = gcc
CFLAGS = -Wall -Wextra -Werror
LDFLAGS =
EXEC = hello

all: $(EXEC)

hello: 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

fclean: clean
    rm -rf $(EXEC)

re: fclean all
```

Step 4 - Les règles d'inférence

Part 4.1 - Une règle d'inférence générique

La syntaxe des Makefiles nous permet de déclarer des règles génériques, telles qu'une règle qui permet de définir la construction d'un fichier .o depuis un fichier .c :

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

Il est alors possible de simplifier notre Makefile :

```
CC = gcc
CFLAGS = -Wall -Wextra -Werror
LDFLAGS =
EXEC = hello

all: $(EXEC)

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

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all
```

PART 4.2 - LA GESTION DE DÉPENDENCES

Le problème de cette version est que le fichier main.o n'est plus reconstruit si le fichier hello.h est plus récent. Il est alors possible de faire fonctionner notre règle d'inférence avec une règle permettant de spécifier la dépendance entre ces deux fichiers :

```
# Makefile

CC = gcc
CFLAGS = -Wall -Wextra -Werror
LDFLAGS =
EXEC = hello

all: $(EXEC)

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

main.o: hello.h

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all
```

STEP 5 - LE .PHONY

En parlant de dépendance, que se passerait-il si un fichier ou un dossier nommé clean se trouvait au même endroit que notre Makefile ? Et bien, la règle clean n'ayant pas de dépendance, le fichier ou le dossier serait considéré comme le plus récent et la règle ne serait jamais exécutée.
Pour pallier à ce genre de problème, il existe la cible .PHONY. Les règles précisées comme dépendances de celle-ci seront exécutées de manières inconditionnelles, peu importe alors si un fichier existe avec le même nom.

```
# Makefile

CC = gcc
CFLAGS = -Wall -Wextra -Werror
LDFLAGS =
EXEC = hello

all: $(EXEC)

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

main.o: hello.h

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all

.PHONY: clean fclean re
```

STEP 6 - LA CONSTRUCTION DES FICHIERS OBJETS

Lors de la réalisation de projets de plus grande taille, on peut rapidement se retrouver avec de nombreux fichiers, il devient alors fastidieux de tous les lister dans la définition de nos règles de compilation. On va alors utiliser d'autres variables afin de résoudre ce problème :

- La variable SRC qui contiendra la liste de tous les fichiers source du projet.
- La variable OBJ qui contiendra la liste des fichiers objet.

La variable SRC se définit de manière assez simple :

```
SRC = hello.c main.c
```

Et si l'on réfléchit bien, le contenu de la variable OBJ est presque le même, à ceci près que, les fichiers se termineront en .o au lieu de .c. Or, il existe une syntaxe qui permet de faire cette conversion à partir de la variable SRC :

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

Voilà donc à quoi ressemble noter Makefile avec ces deux nouvelles variables :

```
# Makefile

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

all: $(EXEC)

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

main.o: hello.h

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all

.PHONY: clean fclean re
```

STEP 7 - L'UTILISATION DE CONDITIONS

Lors de la phase de développement d'un projet, il est fortement recommandé d'utiliser les symboles de débogage pour pouvoir tester efficacement son projet. Or, il ne faut pas que ceux-ci soient présents lors de la mise en production. Pour ce faire, on peut ajouter des conditions à notre Makefile, pour que l'on puisse préciser si on est en phase de développement ou non :

```
# Makefile

DEBUG = yes

CC = gcc
ifeq ($(DEBUG), yes)
    CFLAGS = -Wall -Wextra -Werror -g3
    LDFLAGS =
else
    CFLAGS = -Wall -Wextra -Werror
    LDFLAGS =
endif
SRC = hello.c main.c
OBJ = $(SRC:.c=.o)
EXEC = hello

all: $(EXEC)
ifeq ($(DEBUG), yes)
    @echo "Generation en mode debug"
else
    @echo "Generation en mode release"
endif

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

main.o: hello.h

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all

.PHONY: clean fclean re
```

Ainsi, plutôt que de modifier le Makefile à chaque fois, il suffit de modifier la variable DEBUG.

Part 3 - Aller plus loin

Step 1 - Les sous-makefile

Plus les projets deviennent conséquents, plus il est conseillé de les subdiviser en plusieurs parties. Il n'est donc pas rare de devoir compiler plusieurs parties d'un même projet pour qu'il fonctionne. Pour ce faire, et au lieu d'appeler plusieurs Makefile manuellement, il est préférable de créer un Makefile "maître" qui se chargera d'appeler les autres Makefile du projet. Voici un exemple :

```
# Makefile maitre

HELLO_DIR = hello
EXEC = $(HELLO_DIR)/hello

all: $(EXEC)

$(EXEC):
    $(MAKE) -C $(HELLO_DIR)

clean:
    $(MAKE) -C $(HELLO_DIR) $@

fclean: clean
    $(MAKE) -C $(HELLO_DIR) $@

re:
    $(MAKE) -C $(HELLO_DIR) $@

.PHONY: clean fclean re $(EXEC)
```

```
# Makefile

DEBUG = yes

CC = gcc
ifeq ($(DEBUG), yes)
    CFLAGS = -Wall -Wextra -Werror -g3
    LDFLAGS =
else
    CFLAGS = -Wall -Wextra -Werror
    LDFLAGS =
endif
SRC = hello.c main.c
OBJ = $(SRC:.c=.o)
EXEC = hello

all: $(EXEC)
ifeq ($(DEBUG), yes)
    @echo "Generation en mode debug"
else
    @echo "Generation en mode release"
endif

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

main.o: hello.h

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

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all

.PHONY: clean fclean re
```

Step 2 - La gestion des dépendances

Comme précisé précédemment, il est possible de créer des dépendances manuellement entre les fichiers sources et les header files. Mais, en c++ il arrive de travailler avec beaucoup de header files, il devient alors fastidieux d'écrire toutes les dépendances à la main. Heureusement, il existe une solution pour forcer la génération de ces dépendances lors de la compilation. Pour ce faire, il suffit de rajouter le flag `-MD` dans la règle d'inférence qui définit le passage des fichiers .cpp en .o et d'include les fichiers `.d` qui seront générés. Ce sont ces fichiers qui contiendront les informations nécessaires à la résolution des dépendances.

```
# Makefile

CC = g++
CXXFLAGS = -Wall -Wextra -Werror -std=c++20
LDFLAGS =

SRC = hello.cpp main.cpp
OBJ = $(SRC:.cpp=.o)

EXEC = hello

all: $(EXEC)

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

%.o: %.cpp
    $(CC) -o $@ -MD -c $< $(CXXFLAGS)

clean:
    rm -rf *.o

fclean: clean
    rm -rf $(EXEC)

re: fclean all

-include $(OBJ:%.o=%.d)

.PHONY: clean fclean re
```

Step 3 - Arborescence de fichier (String Substitution)

Maintenant que nous avons vu comment générer les fichiers .d, il devient difficile de se repérer dans les fichiers sources qui sont mélangés avec les fichiers générés lors de la compilation. Pour résoudre ce problème, on va pouvoir se servir des règles de `string substitution`. En effet, celles-ci vont nous être utiles pour générer une copy de l'arborescence des fichiers sources pour les fichiers de compilation.

```
# Makefile

CC = g++
CXXFLAGS = -Wall -Wextra -Werror -std=c++20 -I ./include
LDFLAGS =

SRC_DIR = ./src
SRC =   $(SRC_DIR)/hello.cpp
        $(SRC_DIR)/main.cpp

BUILDDIR = ./obj
OBJ = $(patsubst $(SRC_DIR)/%.cpp,$(BUILDDIR)/%.o,$(SRC))

EXEC = hello

$(BUILDDIR)/%.o:    $(SRC_DIR)/%.cpp
    @mkdir -p $(dir $@)
    $(CC) -o $@ -MD -c $< $(CXXFLAGS)

all: $(EXEC)

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

clean:
    rm -rf $(BUILDDIR)

fclean: clean
    rm -rf $(EXEC)

re: fclean all

-include $(OBJ:%.o=%.d)

.PHONY: clean fclean re
```

Ici, on utilise la fonction de string substitution `patsubst` afin de créer la copie de l'architecture des fichiers source.
Cette fonction respecte le pattern suivant :

```
$(patsubst pattern,replacement,text)
```