TP6 - Makefile

Dans cette planche de TP nous allons approfondir les notions de MakeFile.

Exercice 1. *Makefile*

Exemple de compilation séparée :

hello.c

```c
#include <stdio.h>


void Hello(void){
  printf("Hello World\n");
}
```

hello.h

```c
#ifndef H_GL_HELLO
#define H_GL_HELLO

void Hello(void);

#endif
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"

int main(){
  Hello();
  return EXIT_SUCCESS;
}
```

Structure d'un Makefile :

```
cible: dependance
  commandes

hello.o: hello.c
  gcc -o hello.o -c hello.c -Wall -W -pedentic -ansi
```

Exercice 1. Ecrire un $Makefile$ pour compiler le projet.

Il existe des conventions pour certaines règles :

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

clean : elle permet de supprimer tous les fichiers intermédiaires.

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

Exercice 2. Enrichir votre Makefile précédent en ajoutant les nouvelles cibles.

Il existe aussi des variables dans un $Makefile$. Pour les définirs on écrit au début du fichier :

```
CFLAGS=-W -Wall -ansi -pedantic
```

Comme pour le shell le $\$(CFLAGS)$ sera remplacé par sa valeur.

Exercice 3. Modifier votre $Makefile$ pour utiliser des variables.

indication vous pouvez aussi utiliser des variables pour définir des commandes à lancer (ex : $CC = gcc$)
Vous pouvez faire des règles génériques grâce aux variables spéciales suivantes :

$@	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

Exercice 4. Modifier votre $Makefile$ pour y inclure ces nouvelles variables.

Makefile permet également de créer des règles génériques (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 : Exemple :

```
1 %.o: %.c
2   command
```

Exercice 5. Modifier votre $Makefile$ pour y remplacer les deux règles utilisant les mêmes commandes par une seule règle générique.

Exercice 6. Est-il possible de modifier les autres règles pour qu'il n'y apparaisse plus de nom de fichiers appartenant au projet.

Jusqu'à maintenant, 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.

Exercice 7. Ajoutez la règle .PHONY dans votre $Makefile$

Jusqu'à présent nous listons les fichiers objets (.o). Il est possible de lister les fichiers sources et de remplacer l'extensions de chaque élément de la liste avec :

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

Vous pouvez aussi générer une liste de fichier présent dans votre répertoire grâce à la commande suivante :

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

Exercice 8. Ajoutez les deux dernière fonctionnalités à votre $Makefile$.

Félicitations ! Vous avez un fichier $Makefile$ générique pour presque tous vos projets.

Allons plus loin ! Les Makefiles nous offrent en plus une certaine souplesse en introduisant des 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 tantôt en mode debug, tantôt en mode release sans avoir à modifier plusieurs lignes du Makefile pour passer d'un mode à l'autre. Il suffit de créer une variable DEBUG et tester sa valeur pour changer de mode :

```
DEBUG=yes
ifeq ($(DEBUG),yes)
  CFLAGS=-W -Wall -ansi -pedantic -g
  LDFLAGS=
else
  CFLAGS=-W -Wall -ansi -pedantic
  LDFLAGS=
endif
...
all: $(EXEC)
ifeq ($(DEBUG),yes)
  @echo "generation␣en␣mode␣debug"
else
  @echo "generation␣en␣mode␣release"
endif
```

Dans ce cas, l'exécutable est généré en mode debug, il suffit de changer la valeur de la variable DEBUG pour revenir en mode release.

Exercice 9. Comprenez juste ce code.

Il est possible d'appeler un Makefile depuis un autre Makefile grâce à la variable $(MAKE) et de fournir à ce second Makefile des variables définies dans le premier en exportant celles-ci via l'instruction export, avant d'invoquer le second Makefile.
On peut même lancer des commandes du shell grace à la commande shell. La fonction filter-out permet d'enlever certains éléments d'une liste : en particulier, ici, on ne veut ni le répertoire courrant, ni le répertoire Template.

```
export CC=gcc
export CFLAGS=-W -Wall -ansi -pedantic
export LDFLAGS=
DIR = $(filter-out . ./Template,$(shell find . -type d -print))
all: $(DIR)

clean: $(DIR)

.PHONY: $(DIR)
$(DIR):
        $(MAKE) -C $@ $(MAKECMDGOALS)
```

Exercice 10. Expliquez ce que fait ce code.