Gestion des dépendances

Exemple :

```
# toute ligne qui commence par " # " est un commentaire

# gestion des dépendances
# conversion dépend de conversion.c de sorties.c et de entrees.c

conversion : conversion.o entrees.o sorties.o

# édition de lien : lors de la compilation, la dernière étape consistera  à lier les binaires
# ce n'est pas ld qui sera appelée, mais gcc
# (qui gère directement de nombreuses options à ce stade de la compilation )

# noter les deux tabulations qui doivent précéder toutes lignes  de commande shell
		gcc -o conversion conversion.o entrees.o sorties.o
```

Dépendances secondaires :

Pour spécifier des dépendances secondaires dans un Makefile,
on définira autant de cibles que de fichiers objets,
en précisant leurs dépendances respectives

On reprend les règles précédentes,
en décrivant précisément les dépendances pour chaque .o :

```
conversion : conversion.o entrees.o sorties.o
		gcc -o conversion conversion.o entrees.o sorties.o

conversion.o: conversion.c
		gcc -c conversion.c

entrees.o : entrees.c fichier_en_tetes.h
		gcc-c  entrees.c

sorties.o : sorties.c
	gcc -c sorties.c
```

Règle " clean "

Pour supprimer tous les produits de compilation, on utilise la règle clean
(quelquefois disctclean, ou encore realclean)

Exemple :

```
clean :
        rm -f *.o *.do *~ conversion conversion.db
```

Remarque : on peut aussi créer une règle permettant d'installer l'application :

```
install :
        sudo cp conversion /usr/local/bin
```

ou la désinstaller:

```
uninstall :
        sudo rm -rf /usr/local/bin/conversion
```

En résumé :

Une ligne qui commence par # est un commentaire

ATTENTION : Insérer un # au milieu d'une ligne ne produit pas forcément un commentaire après le signe ' # '

Une commande shell commence après des tabulations (deux tabulations en général)

Une ligne de commande shell crée un shell, donc deux lignes différentes
n'auront pas le même environnement

Exécution de plusieurs commandes dans un même shell :

Solution1 :

```
cd ../programme  ; gcc -c -o truc
```

Solution2 :

```
cd ../programme \
gcc -c -o truc
```

Remarque : le caractère " \ " est utilisé pour couper une ligne trop longue.
Il ne doit être suivi d'aucun autre caractère sur la ligne pour être valable

Cibles

Une cible de make s'écrit :

<nom_de_cible> : <liste_dependances>

Abréviations et Macros définitions

Abréviations:

Pour simplifier l'écriture des Makefile, on utilise des macros définitions de substitutions :

$@ : symbolise le nom complet de la cible

$* : représente le nom de la cible, sans suffixe

Exemple :

```
conversion: conversion.o entrees.o sorties.o
		gcc -o $@ $*.o entrees.o sorties.o

conversion.db: conversion.do entrees.do sorties.do
		gcc -o $@ $*.do entrees.do sorties.do


entrees.o : entrees.c fichier_en_tete.h
		gcc -c -O $*.c

entrees.do: entrees.c fichier_en_tete.h
		gcc -ggdb -o $@ ...
```

Macros définitions :

<nom> = <macro>

Exemples :

la chaine $(<nom>) est substituée à <nom>,
mais on peut aussi utiliser des variables comme DEPS et DEPSDB

```
DEPS = entrees.o sortie.o conversion.o
DEPSDB = entrees.do sortie.do conversion.do

conversion: $(DEPS)
		gcc -o $@ $(DEPS)

conversion.db:   $(DEPSDB)
		gcc -o $@ $(DEPSDB)
```

On peut ajouter le déboguage avec CFLAGS :

CFLAGS = -DDEBUG -gdwarf-2

Ainsi, la ligne :

gcc -c $(CFLAGS) entrees.c

Important : bien respecter la syntaxe

$(CFLAGS) ou ${CFLAGS}

Recherche des répertoires :

on utilise VPATH, avec la syntaxe :

VPATH = <path1>:<path2>:<path3>

Règles de compilation par défaut

Suffixes :

une ligne qui ne contient rien réinitialise les suffixes pris par défaut par make
.SUFFIXES:

on ajoute de nouvelles règles aux fichiers se terminant par .c .o et .do (pour déboguage)

.SUFFIXES : .c .o .do

suivi de la règle

.c.o:; gcc -c $@ -O $*.c

Important : il ne doit pas y avoir d'espace entre .c et .o , qui sont immédiatement
suivis de " :; " , puis de la commande la plus générique possible.

Ici, un .c donnera un .o par défaut, avec la ligne de commande générique associée

La règle précédente peut s'écrire :

.c.o:; gcc -c -o $@ -O $<

Dans laquelle " $< " symbolise le fichier source sur lequel s'appuie la cible

autre exemple, concernant la version déboguage :

.c.do:; gcc -c -o $@ $(CFLAGS) $<

Note : il n'y a pas d'optimisation lors de la compilation de la version de déboguage
(c'est en général vrai, et cela peut causer des dysfonctionnements inattendus)

Etude d'un makefile complet (d'après Programmer avec les outils GNU, éditions O'Reilly):

```
# cette ligne reinitialise la liste des suffixes
.SUFFIXES:
# nos regles
.SUFFIXES: .c .o  .do
OPTIONDEBOG =  -DDEBUG -g
OPTIONPROG = -Wall -ansi -O2
OBJETSPROG = programme.o autre.o
OBJETSDEBOG = programme.do autre.do
NOM_EXEC = prog
# regle suffixée decrivant la creation d'un fichier " .o "
# (version finale )
.c.o:; gcc  -c -o $@ $(OPTIONPROG) $<

# regle suffixee décrivant la création d'un fichier  " .do "
# version de déboguage
.c.do:; gcc -c -o $@ $(OPTIONDEBOG) $<
# Creation des exécutables
prog: $(OBJETSPROG)
```

```
		gcc -o  $(NOM_EXEC) \
			$(OBJETSPROG)

debog : $(OBJETSDEBOG)
		gcc -o $(NOM_EXEC) $(OBJETSDEBOG)

# liste des fichiers à lier à  la version finale
# (et à recompiler au besoin)

programme.o: programme.c en_tete.h

autre.o : autre.c

# liste des fichiers à lier à  la version de déboguage
# (et à recompiler au besoin )


programme.do: programme.c en_tete.h

autre.do : autre.c

# pour effacer tous les produits de compilation

clean :
		rm *.o *.do *~
		rm $(NOM_EXEC)
```

Exemples d'utilisation :

make programme.o

lance la commande :

gcc -c programme.o -Wall -ansi -O programme.c

de même que :

make autre.o

lancera :

gcc -c autre.o -Wall -ansi autre.c

Invoquer make

make <cible>

Options essentielles de make :

-f : permet de choisir le fichier Makefile (remplace le nom par défaut)

-n : ne fait rien, simule la réalisation des règles. Très utile !

-i : ignore les erreurs

-k : analogue à i, mais force la création de toutes les cibles, à
l'exception de celles qui engendrent une erreur

-q : n'exécute aucune commande, mais vérifie que la cible est à jour (retourne 0 si c'est le cas).
Si la cible n'est pas à jour, une autre valeur est retournée

-s : mode silencieux, ne retourne rien

-j : lance l'exécution simultanée de commandes multiples (machine multi processeurs/cores)

Note : il faut que la machine supporte la charge (-j 3 est raisonnable sur une machine mono utilisateur)

-t : lance make en forçant les mises à jour des fichiers concernés

-d : mode bavard (verbose)

Exemples ...