Pour exécuter une suite de commandes prédéfinies, par exemple la compilation en C/C++, la mise à jour de site web, l'archivage et désarchivage de données,… il est possible de les écrire sous forme d'une liste de "cibles", suivies de leurs "dépendances". Cette description est à écrire dans un document « makefile » ou « Makefile ». La commande « make » (utilitaire GNU make) s'en sert alors pour exécuter les "actions" pointées.

Cette présentation est accompagnée d'un petit exemple simple que vous pouvez copier et tester chez vous : un programme hello.c, sa définition dans hello.h et un programme principal main.c. Voir la partie VII.

I. Syntaxe d'une règle

Le makefile contient :

- les cibles : généralement le nom d'un fichier (sauf lors d'actions systématiques, voir plus loin)
- leurs dépendances : les fichiers dont la cible dépend, les fichiers susceptibles d'être modifiés
- les actions : commandes à exécuter

Il n'est permis qu'une seule règle par cible. (c'est-à--dire que la cible n'apparaît qu'une fois dans le fichier de description).

```
cible: fich_a fich_b ... fich_q
action_1
action_2
...
action_n
```

- Une cible est suivie de « : ».
- Une action commence par « », la touche tabulation.
- Un commentaire est précédé du caractère « # ».

- Lorsqu'une ligne est trop longue, il suffit de mettre un « » juste avant un "return" pour la faire continuer sur la ligne suivante.

Fabrication d'une cible:

Pour fabriquer une cible test, il faut taper la commande « make test » (ou « make », voir plus loin). Sa liste de dépendances est alors parcourue. Chaque fichier s'y trouvant est soit un fichier du répertoire, soit une autre cible. Dans le second cas, le fichier est créé avant de poursuivre la liste. Dès que tous les fichiers de dépendance d'une cible sont créés ou existent, les actions de cette cible sont exécutées.

il est possible de faire *make -j 2 test* ou *make -j 2* pour pouvoir compiler en utilisant deux processeurs si on les a. (-j X : X=nombre de processeurs).

II. Un premier makefile

Pour notre exemple, la cible principale est l'exécutable hello.

makefile

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

hello.o: hello.c
      gcc --o hello.o --c hello.c --W --Wall --ansi --pedantic

main.o: main.c hello.h
      gcc --o main.o --c main.c --W --Wall --ansi --pedantic
```

La cible hello dépend des fichiers objets main.o et hello.o. Comme ils n'existent pas, ils deviennent des cibles du makefile. Ils sont alors créés directement puisqu'ils ne dépendent que de fichiers déjà existants dans le répertoire. Et l'exécutable hello est alors créé.

Un makefile correct:

Lorsque l'on souhaite construire plusieurs exécutables avec un seul makefile, il faut rajouter la règle « all: » et y inscrire la liste des cibles à traiter. La commande « make » considère alors toutes les cibles de cette règle.

Il est convenable de se débarrasser des fichiers intermédiaires comme les fichiers objets créés pour la compilation par exemple. C'est par la règle « clean » que le nettoyage se fait (par convention). Il suffit de taper la commande « make clean ».

Enfin, lorsque l'on souhaite tout recompiler, on choisit une règle « again » par exemple et on appel la commande « make again ».

Attention, il ne doit donc pas y avoir de fichiers clean, all et again dans le répertoire.

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 --W --Wall --ansi --pedantic

main.o: main.c hello.h
	gcc --o main.o --c main.c --W --Wall --ansi --pedantic

clean:
	rm --f *.o

again: clean
	make hello
```

III. Les macros générales

A placer en tête du makefile, les macros servent à simplifier l'écriture des commandes, et leur modification. Sous la forme NOM = chaîne de caractère, tous les appels suivants de $(NOM) -ou encore ${NOM}- seront remplacés par chaîne de caractère.

Les macros utilisés par convention sont : CC pour le compilateur, CFLAGS pour les options de compilation, EXEC le nom des exécutables, …

makefile

```
CC = gcc
CFLAGS = --W --Wall --ansi --pedantic
EXEC = hello

all: ${EXEC}

hello: hello.o main.o
	${CC} --o hello hello.o main.o

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 --f *.o

again: clean
	make ${EXEC}
```

IV. Les macros internes

Dans la plupart des cas, les cibles et les dépendances reprennent les mêmes noms de fichier

avec des extensions différentes. Pour ne pas réécrire ces noms, il existe des variables internes :

- > Le nom de la cible est « @ »
- > Le nom de la première dépendance est « - > La liste des dépendances est « ^ »
- > Le nom du fichier sans suffixe est « * »
>...

Attention, ce sont des macros, donc leurs appels sont précédés de « $ ».

makefile

```
   CC = gcc
CFLAGS = --W --Wall --ansi --pedantic
EXEC = hello

all: ${EXEC}

hello: hello.o main.o
	${CC} --o $@ $^

hello.o: hello.c
	${CC} --o $@ --c $
main.o: main.c hello.h
	${CC} --o $@ --c $
clean:
	rm --f *.o

again: clean
	make ${EXEC}
```

V. Les règles d'inférences

Les règles implicites ou règles de suffixes permettent de définir les actions à effectuer pour créer un fichier avec un certain suffixe à partir d'un autre fichier avec un autre suffixe. Dans notre exemples, on peut se servir de la règle : « %.o: %.c » pour les cibles main.o et hello.o.

makefile

```
  CC = gcc
CFLAGS = --W --Wall --ansi --pedantic
EXEC = hello

all: ${EXEC}

hello: hello.o main.o
	${CC} --o $@ $^ ${CFLAGS}

main.o: hello.h

%.o: %.c
	${CC} --o $@ --c $
clean:
	rm --f *.o

again: clean
	make ${EXEC}
```

Attention, comme la cible main.o dépend du fichier hello.h, il faut le rajouter.

VI. Les options

- parallèlisme

```
$ make -j 2
```

L'option "-j X" partage le travail vers les X processeurs de l'ordinateur. (donc "-j 2" pour les double coeurs !)

- make 2&>

```
$ make  2&> .txt
```

L'option 2&> copie les erreurs lors de l'éxécution dans le fichier.

VII. Codes d'un programme C

Petit exemple très simple de programme C pour illustrer ce tutorial.

hello.c

```
 #include
#include

void Hello(void)
 {
    printf("Hello Worldn");
 }
```

hello.h

```
 #ifndef H_HELLO
#define H_HELLO

void Hello(void);

#endif
```

main.c

```
 #include
#include
#include "hello.h"

int main(void)
 {
```

```
    Hello();
    return 0;
}
```

Si vous n'êtes pas à l'aise avec les bases du code C ou de la commande gcc/g++, allez voir leur présentation ici .