Chapitre 1

Utilisation de `make`

1.1 `make` : pour quoi faire ?

`make` est un programme qui détermine automatiquement quelles parties d'un programme doivent être recompilées, et exécute les commandes nécessaires à leur recompilation.

Plus généralement, `make` peut être utilisé pour mettre à jour automatiquement des fichiers quand ceux dont ils dépendent sont modifiés. Les dépendances entre fichiers sont décrites dans un fichier *makefile*. Ce fichier peut s'appeler makefile ou Makefile, il sera alors trouvé automatiquement par `make` (makefile sera trouvé avant Makefile). S'il porte un autre nom, il faudra l'indiquer à `make` grâce à l'option `-f`.

1.2 Introduction aux *makefiles*

Un *makefile* est un fichier qui contient des règles de mise à jour : chaque règle indique que certains fichiers – les cibles – dépendent d'autres fichiers – les dépendances – et qu'il faut exécuter certaines commandes pour mettre à jour les cibles lorsqu'elles sont plus anciennes qu'une de leur dépendances.
Par exemple :

```
prog : main.o demo.o util.o
  gcc -o prog main.o demo.o util.o
```

indique que `prog` est une cible qui dépend de `main.o`, `demo.o` et `util.o`, et que lorsque `prog` est plus ancien qu'une de ces dépendances, il faut le mettre à jour en exécutant la commande indiquée sur la deuxième ligne.

Plusieurs commandes peuvent apparaître dans une règle, chacune sur sa propre ligne. Chaque ligne de commande doit débuter par une tabulation, et les lignes de commande doivent être consécutives (pas de ligne ne commençant pas par une tabulation entre deux lignes de commandes de la même règle. Les lignes qui commencent par # sont des lignes de commentaires.

Pour mettre à jour une cible, il suffit de la passer en argument à `make`. Si on utilise `make` sans argument, il met à jour la première cible du *makefile*.

1.2.1 Variables

Pour alléger l'écriture des *makefiles*, il est possible de définir des variables. Ainsi, dans l'exemple précédent, on peut définir une variable `objets` dont la valeur est la liste des fichiers objets composant le programme :

```
objets = main.o demo.o util.o

prog : $(objets)
  gcc -o prog $(objets)
```

Le nom de la variable peut être placé entre parenthèses : `$(variable)`, ou entre accolades : `${variable}`. Le symbole $ étant utilisé pour obtenir la valeur d'une variable, on en utilisera deux ($$) pour obtenir un $ dans une règle.

Il est possible de substituer une sous-chaîne à une autre lors de l'expansion d'une variable. Par exemple, `$(var:%c=%o)` donne la valeur de la variable `var` dans laquelle les mots formés d'une suite quelconque de caractères (représentée par le signe `%` suivie des caractères `.c` sont remplacés par la même suite de caractères suivie de `.o`. On peut ainsi écrire notre *makefile* sous la forme suivante qui déduit automatiquement la liste des fichiers objets de celle des fichiers sources :

```
# Liste des fichiers sources
sources = main.c demo.c util.c
# Liste des fichiers objets déduite de celle des sources
objets = $(sources:%.c=%.o)

# 1ere règle = règle par défaut
prog : $(objets)
  gcc -o $@ $(objets)

main.o : main.c demo.h util.h
  gcc -c main.c

demo.o : demo.c demo.h util.h
  gcc -c demo.c

util.o : util.c util.h
  gcc -c util.c
```

On utilise ici la variable prédéfinie `$@` dont la valeur est le nom de la cible. Nous verrons d'autres variables prédéfinies dans le cadre des motifs de règles.

1.2.2 Motifs de règles

Dans l'exemple précédent, la répétition des règles de mise à jour des fichiers objets est fastidieuse. On peut exprimer ces règles qui se ressemblent grâce à un motif de règles :

```
%.o : %.c
  gcc -c $< -o $@
```

Ce motif indique à `make` qu'un fichier d'extension .o dépend d'un fichier de même radical mais d'extension .c et que la mise à jour s'effectue grâce à la commande indiquée. Cette commande utilise deux des variables prédéfinies par `make` :

- `$@` contient le nom de la cible
- `$<` contient le nom de la première dépendance
- `$?` contient le nom des dépendances qui sont plus récentes que la cible
- `$*` dans le cadre d'un motif de règle, contient la partie du nom de la cible qui correspond au symbole %.

L'inconvénient du motif de règles dans cet exemple est que les dépendances sur les fichiers d'entête n'y figurent pas. On peut utiliser une règle sans commande pour les exprimer. Notre *makefile* devient alors :

```
sources = main.c demo.c util.c
objets = $(sources:%.c=%.o)

prog : $(objets)
  gcc -o $@ $(objets)

%.o : %.c
  gcc -c $< -o $@

main.o : main.c demo.h util.h
demo.o : demo.c demo.h util.h
util.o : util.c util.h
```

1.2.3 Inclusion de *makefiles*

Dans la pratique, il est assez difficile de maintenir à jour les dépendances sur les fichiers d'en-tête. Si on indique des dépendances qui n'existent pas, on provoque des recompilations inutiles, et si on en oublie, le programme risque de ne pas fonctionner correctement.

Les compilateurs C disposent en général d'une option `-M` qui leur fait générer une ligne de dépendance pour chaque fichier passé en argument. Ainsi, `gcc -M main.c` affiche `main.o: main.c demo.h util.h stdio.h`.
Sous Linux, `gcc` accepte de plus une option `-MM` qui ignore les fichiers d'entête système (stdio.h n'est donc pas mentionné dans les dépendances lorsqu'on utilise cette option).

Nous pouvons donc générer de façon fiable les dépendances de nos fichiers objets :

```
depend.make :
  gcc -MM $(sources) > depend.make
```

Cette règle indique que la cible `depend.make` ne dépend de rien (elle est donc toujours à jour si elle existe), et qu'elle se met à jour en demandant à `gcc` de générer les dépendances des fichiers sources dans un fichier nommé depend.make (le symbole > indique que le résultat de l'exécution de la commande doit être placé dans un fichier au lieu d'être affiché).

Le fichier depend.make doit être mis à jour à chaque fois que les dépendances sur les fichiers d'en-tête changent. Le problème est qu'il n'est pas facile de détecter automatiquement ces changements, et nous avons donc pris le parti

d'indiquer manuellement que les dépendances ont changé, comme nous le verrons en 1.3.3. Une autre solution consiste à faire dépendre depend.make de tous les fichiers sources, en-têtes y compris. Mais ceci provoque la mise à jour de depend.make à chaque modification d'un fichier source, même si cette modification ne modifie pas les dépendances.

Il ne reste qu'à inclure les règles générées par gcc dans notre *makefile* pour que `make` exploite un jeu de règles complet et à jour :

```
sources = main.c demo.c util.c
objets = $(sources:%.c=%.o)

prog : $(objets)
  gcc -o $@ $(objets)

%.o : %.c
  gcc -c $< -o $@

depend.make :
  gcc -MM $(sources) > depend.make

include depend.make
```

Lors de la première utilisation de ce makefile, `make` ne trouvera pas le fichier depend.make qu'on lui demande d'inclure. Il utilisera la règle dont depend.make est la cible pour créer ce fichier. Le *makefile* sera alors complet et à jour.
Remarque : Ce comportement n'est pas celui de toutes les versions de `make`. Certaines versions ne cherchent pas à générer un fichier inclus absent, il faut alors utiliser une solution plus complexe.

1.3 Idiomes usuels

Certaines règles ou certains types de règles se rencontrent fréquemment dans les *makefiles*.

1.3.1 Cible par défaut

La première règle du *makefile* indiquant la cible à mettre à jour par défaut, on place généralement en tête d'un *makefile* soit une règle mettant à jour le programme complet, soit une règle affichant le mode d'emploi du *makefile*. Dans le premier cas, il s'agit d'une règle sans commande :

```
tout : prog1 prog2
```

qui indique que par défaut, on met tout à jour, c'est-à-dire `prog1` et `prog2` (qui sont les cibles d'autres règles).

Dans le second cas, on a une règle sans dépendance, mais avec commandes :

```
defaut :
  @echo "Compilation et test de Prog"
  @echo "  - tapez 'make prog' pour compiler Prog"
  @echo "  - tapez 'make test' pour tester Prog"
```

Les `@` en tête des lignes de commande indiquent à `make` de ne pas afficher les commandes avant de les exécuter. La commande `echo` affiche son argument, il serait donc pénible de voir cet argument deux fois si `make` affichait ce qu'il va exécuter.

1.3.2 Cibles nulles

Lorsqu'une cible n'a ni dépendances ni commandes et qu'il ne s'agit pas d'un fichier existant, `make` considère qu'elle est mise à jour dès que sa règle est appliquée. On obtient ainsi une cible qui est toujours plus récente qu'une cible qui dépend d'elle. On peut alors utiliser cette cible pour provoquer systématiquement l'exécution de commandes. On emploie généralement le nom `FORCE` pour une telle cible :

```
FORCE:

clean : FORCE
  -rm -f $(objets)
  -rm -f core
```

La commande `make clean` permettra de supprimer les fichiers objets ainsi qu'un éventuel fichier `core` créé lors d'un plantage du programme. Le signe `-` placé en tête d'une ligne de commande indique à make d'ignorer une éventuelle erreur lors de l'exécution de la commande. Cela permet ici de supprimer un fichier `core` même si la suppression des fichiers objets échoue (par exemple s'il en manque certains).

1.3.3 Cibles « bidons »

Il est fréquent d'utiliser des cibles qui ne correspondent pas à un fichier. On parle alors de cibles « bidons » ou « ergodiques » (faites votre choix). La cible `clean` du paragraphe précédent en est un bon exemple.

Ces cibles correspondent à une action que l'on veut effectuer : supprimer les fichiers inutiles, mettre à jour les dépendances etc. Au lieu de faire dépendre ces cibles d'une cible vide du type `FORCE`, on peut indiquer à `make` qu'il s'agit de cibles bidons grâce à la directive `.PHONY` lorsqu'on utilise le programme `make` de GNU (ce qui est le cas sous Linux). L'intérêt de cette directive est qu'elle permet au *makefile* de fonctionner correctement même si un fichier de même nom qu'une cible bidon existe par accident. Ainsi :

```
.PHONY : clean
clean :
  -rm -f $(objets)
  -rm -f core
```

est équivalent, avec GNU `make` à :

```
clean : FORCE
  -rm -f $(objets)
  -rm -f core

FORCE :
```

qui ne fonctionne que s'il n'existe pas de fichier nommé FORCE. Parmi les cibles ergodiques classiques, on trouve, en plus de `clean` :

`clobber` qui supprime tous les fichiers qui peuvent être reconstruits, alors que `clean` préserve traditionnellement les exécutables, les fichiers de configuration etc. On fait habituellement dépendre `clobber` de `clean` puisqu'elle fait un ménage plus poussé.

`depend` qui reconstruit le *makefile* depend.make qui contient les dépendances sur les fichiers d'entête. En fait, cette cible ne fait que détruire le fichier depend.make qui sera reconstruit lors de la prochaine invocation de make.

```
clobber : clean
  -rm -f prog

depend : FORCE
  -rm -f depend.make
```

1.3.4 Cibles datées

Cette catégorie de cibles utilise un fichier vide pour mémoriser la date d'un événement. Par exemple, pour imprimer tous les fichiers sources modifiés depuis la dernière impression, on pourra utiliser :

```
print : $(sources)
  for fich in $? ; do cprint $$fich | lpr -J $$fich ; done
  touch print
```

La date de modification du fichier print est modifiée grâce à la commande `touch` à chaque fois que des fichiers sont imprimés. Ce fichier mémorise donc la date de la dernière impression.

Lorsqu'on exécute `make print`, `make` détermine si certains fichiers sources sont plus récents que le fichier print ou non. Si c'est la cas, la variable `$?` prend pour valeur la liste de ces fichiers, et le corps de la boucle `for` est exécuté pour chacun d'eux et utilise la commande `cprint` pour formater le code, et la commande `lpr` pour l'envoyer à l'imprimante (l'option `-J` indique le nom du fichier à `lpr` pour éviter qu'il n'affiche stdin sur la page de garde). Le double `$` devant `fich` devient un simple `$` quand `make` passe la commande au shell, c'est donc bien la variable `fich` de la boucle `for` qui est utilisée, et non la variable `fich` du *makefile*.

1.4 Exemple de Makefile

Voici un exemple de *makefile* que vous pouvez utiliser en modifiant la valeur des trois premières variables : `programme`, `sources` et `entetes`.

```
###
# Définitions
###
# Nom du programme, cible par défaut
programme = projet
# Fichiers sources (tous les fichiers C du répertoire courant)
```

```
sources = main.c demo.c util.c
# Fichiers d'en-tête (utilisé par la cible print)
entetes = demo.h util.h
# Fichiers objets, déduit de $(sources)
objets = $(sources:%.c=%.o)
# Commande d'édition de liens
link = gcc
# Bibliothèques utilisées par le programme (ici, Xgraphics
# et toutes les bibliothèques qu'elle utilise)
biblio = -lXg -lXpm -lXt -lXmu -lXext -lX11 -lm
# Option de mise au point
debug = -g
# Compilateur C
compile = gcc -c
# Options du compilateur C (ici, pas d'arrêt après les messages
# et affichage systématique du nom du fichier)
coptions = -zsam +zafn -ztr
# Commande de génération des dépendances
depend = gcc -MM

# Cible par défaut : mise à jour du programme
$(programme) : $(objets)
  $(link) $(debug) -o $@ $(objets) $(biblio)

# Motif de règles pour compiler les fichiers .c
%.o : %.c
  $(compile) $(debug) $(coptions) $< -o $@

# Impression des fichiers modifiés depuis la dernière impression
print : $(sources) $(entetes)
  for fich in $? ; do cprint $$fich | lpr -J $$fich ; done
  touch print

# Impression de tous les fichiers
printall : FORCE
  for fich in $(sources) $(entetes) ; do cprint $$fich | lpr -J $$fich ; done

###
# Cibles ergodiques (aussi dites "bidon")
###
# Règle de génération du fichier de dépendances
depend.make :
  $(depend) $(sources) > depend.make

# Règle pour forcer la mise à jour des dépendances
depend : FORCE
  -rm -f depend.make

# Nettoyage modéré
clean : FORCE
```

```
    -rm -f $(objets)
    -rm -f core

  # Nettoyage violent
  clobber : clean depend
    -rm -f prog

  # Cible nulle permettant de forcer l'exécution des commandes
  # de mise à jour des cibles ergodiques
  FORCE :

  # Inclusion du fichier de dépendances
  include depend.make
```

1.5 Mais encore...

Pour plus d'informations sur les options de `make`, consultez la documentation (`man make` ou `kdehelp`).

Pour la documentation complète de `make` (version GNU), consultez le manuel détaillé au format *info* (`info 'GNU make'` ou `kdehelp`), ou au format HTML ou PDF dans les répertoires /usr/doc, /usr/local/doc, /usr/share/doc ou /usr/local/share/doc.