Introduction à la compilation séparée avec GNU Make

Yann Régis-Gianas

4 février 2015

1 Introduction

Make est un utilitaire de gestion des dépendances entre fichiers. Il est souvent utilisé pour la compilation car un projet est formé d'unités (ou modules) qui dépendent les unes des autres de manière hiérarchique. On l'utilise aussi pour automatiser des tâches répétitives comme la génération de documentation, le lancement de batteries de tests, le nettoyage de fichiers sources ou bien la création de l'archive tar (*tarball* en anglais) ou zip de la distribution. Pour l'installation d'un logiciel à partir de ses sources sous Unix, on utilise souvent la commande `make && make install`.

Make est un outil de développement standard sous Unix, normalisé Posix.2. Il en existe de nombreuses versions qui fournissent des extensions à cette norme. Nous utiliserons GNU Make, la plus populaire, mais nous resterons dans le cadre du standard Posix.

La description des dépendances entre fichiers est faite par le développeur sous la forme d'un fichier `Makefile`. L'objectif de ce TP est d'écrire un makefile pour un projet C.

Note : toutes les lignes commençant par ▷ correspondent à des actions que vous devez effectuer pour la bonne marche du TP.

2 Un premier makefile

▷ À la racine de votre compte, décompressez l'archive :

```
tar xfz tp-make.tgz
```

▷ Éditez le fichier `~/TP-Make/Projet/src/Makefile` dont le contenu est :

```
# déclaration de variables
CC = gcc
CFLAGS = -Wall

# règles
all: main
```

```
compute_result.o: compute_result.c compute_result.h \
                  compute_struct.h tools.h
        $(CC) $(CFLAGS) -c compute_result.c

parse_num.o: parse_num.c parse_num.h tools.h
        $(CC) $(CFLAGS) -c parse_num.c

print_result.o: print_result.c print_result.h compute_struct.h \
                tools.h
        $(CC) $(CFLAGS) -c print_result.c

main.o: main.c print_result.h compute_result.h parse_num.h
        $(CC) $(CFLAGS) -c main.c

main: main.o print_result.o parse_num.o compute_result.o
       $(CC) -o main print_result.o parse_num.o \
                     compute_result.o main.o

.PHONY: re clean

clean:
        rm --force main *.o *~ \#*

re: clean main
```

Nous allons d'abord comprendre ce makefile et ensuite l'améliorer.

3 Compiler avec Make

Un makefile simple est composé de déclarations de variables et de règles. Nous rentrerons dans le détail du format de ces différents composants un peu plus tard. Les lignes commencant par un # sont des commentaires.

▷ Identifiez la catégorie des différentes lignes de ce makefile.
▷ En supposant que vous avez ouvert l'archive à la racine, saisissez

```
~$ cd TP-Make/Projet/src
~/TP-Make/Projet/src$ make
```

La compilation du projet s'effectue. Vous pouvez lancer l'exécutable, c'est un programme qui effectue la division euclidienne d'un nombre n donné en argument par 3 (il retourne p et q tels que $n = 3 \times p + q$).

▷ Par exemple :

```
~/TP-Make/Projet/src$ ./main 26
8 2
```

3.1 L'importance des règles

3.2 Pour maintenir les fichiers à jour

Le makefile est une base de données de dépendances entre fichiers. Par exemple, on voit que les fichiers `main.o`, `print_result.o` et `compute_result.o` dépendent du fichier `compute_struct.h`. En effet, ce fichier définit une structure de données qui est utilisée lors du calcul et de l'affichage du résultat.

▷ Modifiez le fichier `compute_struct.h`, en rajoutant une ligne vide par exemple ou bien en lancant la commande :

```
~/TP-Make/Projet/src$ touch compute_struct.h
```

▷ Relancez make :

```
~/TP-Make/Projet/src$ make
```

Le fichier `compute_struct.h` est plus récent que les fichiers qui en dépendent, donc le système recompile les fichiers `print_result.o` et `compute_result.o` mais pas le fichier `parse_num.o` car celui-ci n'utilise pas `compute_result.h`.

3.3 Pour maintenir la cohérence du projet

Spécifier correctement les dépendances d'un projet n'est pas facultatif, par exemple

- ▷ supprimez la dépendance entre le fichier objet `print_result.o` et l'entête `compute_struct.h`,
- ▷ modifiez `compute_struct.h` en échangeant l'ordre des champs de la structure `compute_struct_t`,
- ▷ relancez la compilation :
  ```
  ~/TP-Make/Projet/src$ make
  ```
- ▷ et finalement l'exécution :
  ```
  ~/TP-Make/Projet/src$ ./main 16
  1 5
  ```

La compilation s'est bien passée bien et pourtant le programme est faux !

▷ Vérifiez les dépendances de ce makefile en dessinant le graphe induit par ses règles (à un fichier correspond un nœud, et la relation représentée par la flèche est « dépend de »).

3.4 Le format des règles

Une règle suit la syntaxe suivante :

```
cibles: dependances
[TAB]  commande shell
[TAB]  commande shell
[TAB]  ...
```

Nous avons vu précédemment une cible `.PHONY` dont le nom est prédéfini et la règle est alors traitée spécialement. Nous verrons plus loin sa sémantique.

3.5 Tabulation en début de commande

L'oubli des tabulations au début de chaque ligne de commande est une erreur commune. Dans ce cas-là, Make produit une erreur de la forme :

```
Makefile:13: *** missing separator.  Stop.
```

Pour détecter facilement ce type d'erreur, vérifiez que vous utilisez le mode Emacs idoine (M-x makefile-mode) : les tabulations sont alors colorées.

3.6 Cibles multiples

On constate que les cibles peuvent être multiples. Un même graphe de dépendances peut donc être écrit de manières différentes.

▷ Testez le makefile suivant :

```
CC = gcc
CFLAGS = -Wall

.PHONY: re clean

all: main

parse_num.o print_result.o compute_result.o: tools.h

print_result.o compute_result.o: compute_struct.h

compute_result.o: compute_result.c compute_result.h
	$(CC) $(CFLAGS) -c compute_result.c

parse_num.o: parse_num.c parse_num.h
	$(CC) $(CFLAGS) -c parse_num.c

print_result.o: print_result.c print_result.h
	$(CC) $(CFLAGS) -c print_result.c

main.o: main.c print_result.h compute_result.h parse_num.h
	$(CC) $(CFLAGS) -c main.c

main: main.o print_result.o parse_num.o compute_result.o
	$(CC) -o main print_result.o parse_num.o compute_result.o \
	         main.o

clean:
	rm --force main *.o *~ \#*

re: clean main
```

▷ Vérifiez que ce makefile est bien équivalent au premier en dessinant son graphe de dépendances.

On voit que les définitions des dépendances peuvent se faire de manière indépendantes même si elles portent sur les mêmes fichiers.

3.7 Règle `.PHONY`

La cible prédéfinie ayant pour nom `.PHONY` permet de déclarer des règles qui ne produisent pas de fichiers mais qui doivent être toujours exécutées si on les appelle. Supposons que l'on n'aie pas déclaré de règle `.PHONY: clean`. Lorsque l'on appelle `make clean`, puisqu'il n'y a pas de fichier nommé `clean` Make conclut que cette cible doit être mise à jour et donc envoie la commande correspondante (`rm ...`) à l'interprète de commandes sous-jacent. Mais si, par erreur, un fichier `clean` est créé, lorsque l'on appelle `make clean`, puisqu'il n'y a pas de dépendances associées, Make conlut que le fichier est à jour et donc *la commande n'est pas effectuée*. La règle `.PHONY` force Make a effectuer la commande même si la cible est à jour.

3.8 Règles à motif

Les dépendances peuvent suivre une règles plus générales définies à l'aide de motif (*pattern* en anglais). Ainsi, on peut décrire comment construire un fichier objet à partir d'un fichier source :

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

Cette règle signifie que pour construire un fichier objet à partir d'un fichier source, il faut compiler le fichier `.c` par la commande `$(CC) $(CFLAGS) -c $<`. Le `$<` est une variable définie automatiquement pour toute règle et qui symbolise le premier fichier de la liste de dépendances.

▷ Modifiez le makefile en rajoutant cette règle.

Il y a d'autres variables automatiques :

- `$<` : le premier fichier de la liste de dépendances ;
- `$^` : tous les fichiers de la liste de dépendances (en fusionnant les doublons) ;
- `$+` : tous les fichiers de la liste de dépendances (sans fusionner les doublons) ;
- `$*` : le nom sans extension de la cible ;
- `$@` : la cible.

▷ Par exemple, on peut modifier la règle précédente en :

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

▷ De même, la création de l'exécutable (édition de liens) peut s'écrire :

```
main: main.o print_result.o parse_num.o compute_result.o
     $(CC) $(CFLAGS) -o $@ $^
```

3.9 Format des commandes

Les commandes qui sont exécutées par les règles sont en fait du script *shell* UNIX. On peut donc faire énormément de choses au sein des règles.

▷ Modifiez la règle de compilation pour afficher la dernière date de création du fichier cible :

```
%.o : %.c
	@(test -f $@ && \
	echo Last modification of $@ was: \
	`find $@ -printf %a`.) || \
	echo $@ does not exist.
	$(CC) $(CFLAGS) -c $< -o $@
```

Plusieurs choses à noter :

- `@` sert à spécifier à make de ne pas afficher la commande avant de l'exécuter.
- les \ servent à continuer la ligne (comme en *shell*). Cette règle contient donc deux commandes, la première servant à faire l'affichage de la date et la seconde est la compilation à proprement parler.
- les commandes *shell* sont par défaut envoyée à l'interprète de commande UNIX sh. On peut en spécifier un autre à l'aide de la variable prédéfinie de Make nommée `SHELL`.

La dernière utilisation classique des règles est l'automatisation de tâches. On peut créer des règles qui ne dépendent de rien et dont le rôle est d'exécuter une commande *shell*. Par exemple, la règle `clean` du makefile permet de nettoyer le projet en ne laissant que les sources.

▷ On peut appeler ces règles directement par une commande *shell* :

```
~/TP-Make/Projet/src$ make clean
```

En fait, lorsqu'on lance Make sans argument, c'est la première règle trouvée qui est exécutée. Ainsi dans notre cas, la règle `all` est donc exécutée. Comme elle dépend de l'existence du fichier `main`, la compilation du projet est lancée.

La règle `re` du makefile est une erreur commune a ne surtout pas commettre. Souvent, lorsqu'on a mal établi ses dépendances, on a tendance à compiler le projet par la commande : `make clean && make`. C'est exactement ce que fait la règle `re` : recompiler entièrement le projet. Or, cette utilisation de Make remet en cause tous ses avantages, c'est-à-dire la capacité de ne recompiler que le strict minimum. Lors de gros projets, il est essentiel d'avoir bien défini ses dépendances, sinon des temps d'attente de l'ordre de la dizaine de minutes sont à prévoir à chaque compilation !

▷ Supprimez la règle `re` car ce genre de règle est à bannir.

3.10 Variables

On définit une variable suivant le format :

```
NOM=VALEUR
```

On accède au contenu d'une variable en la déréférençant à l'aide du `$`, ainsi `$(NOM)` représente la suite de caractères `VALEUR`.

Les variables permettent par exemple de centraliser les informations qui peuvent être différentes en fonction des systèmes. Par exemple, dans le makefile, on a stocké le nom du compilateur à l'intérieur de la variable `CC`. Ainsi, si on veut utiliser un autre compilateur C, on doit seulement modifier cette valeur et non modifier l'appel un peu partout dans le makefile. Pour rendre les makefiles plus facile à adapter en fonction des systèmes, on a l'habitude de mettre les déclarations de variables en tête du fichier. Les variables peuvent aussi servir à contenir une liste de fichiers pour y faire référence par la suite.

▷ Ainsi, dans le makefile, définissez la variable `OBJECTS` représentant les fichiers objets du projet ainsi que la variable `PROGRAM` définissant le programme exécutable final :

```
PROGRAM=main
OBJECTS=main.o print_result.o parse_num.o compute_result.o
```

▷ La règle de création de l'exécutable devient donc :

```
$(PROGRAM): $(OBJECTS)
	$(CC) -o $@ $+
```

Attention, on pourrait être tenté de définir la variable `OBJECTS` ainsi :

```
OBJECTS=*.o
```

mais ce serait une erreur. En effet, lorsqu'on affecte `*.o` à `OBJECTS`, il n'y a pas d'interprétation de `*.o` *dans les versions récentes de Make* : il n'y a qu'une affectation de chaîne de caractères et `$(OBJECTS)` vaut `"*.o"`. Des versions anciennes de Make interprètent `*.o` comme le ferait le *shell* : s'il existe des fichiers avec l'extension `.o` leurs noms sont expansés, sinon `*.o` devient une chaîne de caractères. Ce comportement a donc disparu dans les versions récentes.

▷ Essayez de définir `OBJECTS` ainsi.

On peut pourtant déduire automatiquement la liste des objets en utilisant les fonctions de manipulation des variables de Make.

▷ Par exemple, la fonction `$(wildcard PATTERN...)` permet d'effectuer une interprétation de motif (ou *pattern*) *shell* au moment de la définition de la variable :

```
SOURCES=$(wildcard *.c)
```

La fonction `$(patsubst PATTERN,REPLACEMENT,TEXT)` permet de remplacer toutes les occurrences filtrées par un motif dans un texte.

▷ Ainsi, définissez :

```
OBJECTS=$(patsubst %.c,%.o, $(SOURCES))
```

ce qui a pour effet de remplacer toutes les extensions `.c` des fichiers apparaissant dans `$(SOURCES)` par `.o`

Il existe aussi `$(subst FROM,TO,TEXT)` qui remplace dans le texte `TEXT` chaque occurrence de `FROM` par `TO`. Pour plus d'informations sur ces fonctions reportez-vous à la documentation de Make.

3.11 Directives

Les directives sont des instructions spéciales qui agissent sur le comportement de Make. Il existe de nombreuses directives, aussi on n'envisagera ici que les plus utiles.

La directive `include` permet d'inclure un makefile dans un autre makefile, dans le but, par exemple, de centraliser les informations de configuration du projet dans un unique fichier. Tous les makefiles du projet l'incluent alors en tête.

▷ Créez un fichier `~/TP-Make/Projet/Makefile.conf` et déplacez-y les définitions de variables dépendantes du projet.

▷ Placez `include ../Makefile.conf` à la place de vos variables dépendantes du projet dans `~/TP-Make/Projet/src/Makefile`.

3.12 Options de Make

Make est un utilitaire qui possède des options. Par exemple, si votre makefile ne se nomme pas `Makefile`, vous pouvez spécifier son nom avec l'option `-f`.

▷ Copiez le fichier `~/TP-Make/Projet/src/Makefile` dans un fichier nommé, par exemple, `MyMakefile`, et utilisez cette option.

```
~/TP-Make/Projet/src$ cp Makefile MyMakefile
~/TP-Make/Projet/src$ make -f MyMakefile
```

Lorsque l'on n'est pas sûr du comportement de Make, on peut le lancer en lui demandant d'imprimer les commandes mais de ne pas les transmettre à l'interprète sous-jacent. Cela se fait avec l'option `-n` (*just print*).

Lorsque l'on ne comprend plus le comportement de Make sur un makefile, on peut le lancer en mode débogage à l'aide de l'option `-d` (*debug*) :

```
~/TP-Make/Projet/src$ make -d
```

On remarque que Make teste beaucoup de choses implicitement. L'utilisation de ce comportement est l'objet de la section suivante.

L'option `-k` (*keep going*), quant à elle, permet de continuer la compilation des fichiers même en cas d'erreur de certaines compilations.

On peut finalement noter l'option `-j` (*jobs*) qui permet de lancer les compilations en parallèle lorsque le matériel le permet. Sur des systèmes multiprocesseurs, on peut ainsi gagner beaucoup de temps.

4 Utilisation courante

Cette section explique une utilisation de Make qui repose sur des variables et des règles définies de manière implicite par des conventions de programmation.

4.1 Règles et variables implicites

Comme la compilation de code C est une tâche très courante sur les systèmes Unix, la règle qui décrit comment compiler un fichier `.c` en `.o` est prédéfinie par Make, comme on a pu le constater sur le makefile initial.

▷ Supprimez la règle de compilation du makefile et lancez :

```
~/TP-Make/Projet/src$ make clean
~/TP-Make/Projet/src$ make
```

La compilation des fichiers sources s'est effectuée sans qu'on ait eu besoin d'écrire la règle ! En fait, la règle prédéfinie correspondante possède la forme suivante :

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

Les variables `CC`, `CPPFLAGS` et `CFLAGS` sont prédéfinies et signifient :

- `CC` : le compilateur C ;
- `CPPFLAGS` : les options du préprocesseur (comme par exemple `-I`) ;
- `CFLAGS` : les options du compilateur (comme `-O2`).

▷ Ainsi, on peut définir ces variables au niveau du *shell* :

```
~/TP-Make/Projet/src$ make clean
~/TP-Make/Projet/src$ make CFLAGS=-O2
```

De la même manière, la création d'exécutable est gérée par défaut par Make.

▷ Supprimez la commande de création de l'exécutable et ensuite :

```
~/TP-Make/Projet/src$ make clean
~/TP-Make/Projet/src$ make
```

En fait, la règle prédéfinie (et implicite) est de la forme :

```
%: %.o
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^
```

La variable `LDFLAGS` correspond aux options pour le programme qui effectue l'édition de liens (usuellement ld). On peut par exemple spécifier ici des options donnant les chemins des bibliothèques, telles que `-Ldirectory` et `-llibrary`.

4.2 Applications récursives

Pour des projets moyens ou grands, il est souvent utile ou nécessaire de hiérarchiser les sources et donc de placer les fichiers dans des répertoires différents (correspondant alors logiquement à des composants du logiciel à construire). Pour gérer la compilation de tels projets, il est possible d'effectuer des appels récursifs de Make dans les sous-répertoires, de telle sorte que chaque composant (ou module) possède son propre makefile et soit compilé séparément.

▷ Créez un makefile à la racine du projet (`~/TP-Make/Projet/Makefile`) pour factoriser l'appel de la règle `clean` dans tous les sous-répertoires du projet. Il contient alors :

```
SUBDIRS = src doc test
clean:
     for dir in $(SUBDIRS); do \
       $(MAKE) clean -C $$dir; \
     done
```

Les deux signes dollar dans `$$dir` servent, le premier, à éviter que Make ne confonde `$dir` avec une variable gérée par lui, et le second est pour l'interprète de commandes.

Cette méthode d'application récursive est très pratique car elle permet de modulariser ses makefiles tout en factorisant leurs informations communes. Vous retrouverez ce principe à l'œuvre dans de nombreux projets.

Néanmoins, de nos jours on critique l'utilisation des appels récursifs de Make et l'option `-C` (*change directory*). On constate en effet que si on a une multitude de makefiles au sein d'un même projet, lorsqu'on lance une commande, par exemple `make all`, le makefile principal fait appel à tous les sous-makefiles pour savoir si les cibles doivent être reconstruites. En effet, il ne connaît pas les dépendances du projet entier donc il peut appeler certains makefiles de sous-répertoires sans que cela soit nécessaire. Ces appels engendrent une certaine lenteur du système car on calcule les dépendances localement à chaque sous-répertoire : les calculer globalement serait plus efficace.

Une autre solution consiste donc à n'utiliser qu'un seul makefile qui inclut des fichiers décrivant les dépendances de chaque module (sous-répertoire) du projet. Dans ces fichiers, on *augmente* les variables du makefile principal.

▷ Définissez une variable `CLEANFILES` dans le makefile principal, c'est-à-dire situé à la racine du projet (`~/TP-Make/Projet/Makefile`).

Dans le fichier de description du sous-répertoire, on trouve maintenant :

```
# The dependences ...
src/main: src/main.o src/print_result.o src/parse_num.o \
          src/compute_result.o
...
# The CLEANFILES of the module.
CLEANFILES += src/main.o src/printer_result.o src/parse_num.o \
              src/compute_result.o
```

Maintenant la règle `clean` s'écrit :

```
clean:
    rm --force $(CLEANFILES)
```

▷ Définissez un fichier `dep.mk` dans chaque sous-répertoire dans lequel vous insérez la partie de makefile nécessaire au module. Attention, il faut désormais résonnez en considérant que vous vous situez à la racine du projet.

▷ Incluez ces makefiles dans le makefile principal et supprimez les appels récursifs.

On évite ainsi les appels récursifs à Make et donc on gagne du temps.

4.3 Génération automatique des dépendances

Il est possible de définir automatiquement la plupart des dépendances d'un projet à l'aide de programmes comme `makedepend` ou `gcc`.

▷ Voici l'exemple fourni dans la documentation de Make :

```
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm --force $@.$$$$
```

L'option `-M` de GCC permet d'obtenir la liste des fichiers inclus par un source C ainsi que ce fichier C lui-même : on obtient ainsi l'ensemble de tous les fichiers dont dépend le fichier objet. On redirige cette sortie vers un fichier temporaire. Ensuite, la commande `sed` insère le fichier de dépendances au sein des fichiers cibles de la règle. En effet, si on modifie le source, par exemple en rajoutant une inclusion, les dépendances doivent être regénérées. Au final, la sortie de SED donne le contenu du fichier de dépendances et on peut effacer le fichier temporaire.

Voici une explication de la commande SED ci-dessus. Tout d'abord, il faut se souvenir que SED (*Stream EDitor*) permet de transformer un flux de caractères. Une des commandes les plus utilisées est la substitution `s, regexp, regexp, g`. Elle effectue un remplacement de tout sous-mot filtré par la première expression régulière par la seconde. Il faut échapper les caractères qui correspondent à la syntaxe des expressions régulières et non au caractère du flux (cela explique les `\(` par exemple). Dans notre cas, on utilise une référence à l'entrée dans la sortie : le `\1` fait référence à ce qui a été capturé par le premier jeu de parenthèses (ici c'est le nom du fichier sans l'extension `.o`). Notre commande `sed` consiste donc à chercher le nom de fichier objet éventuellement suivi d'un « : » et de le remplacer par ce même nom de fichier suivi par le nom du fichier de dépendances (extension `.d`) et, enfin, le « : ». Par exemple

```
parse_num.o : parse_num.c parse_num.h
```

devient

```
parse_num.o parse_num.d : parse_num.c parse_num.h
```

▷ Spécifiez l'inclusion des fichiers ainsi produits à la fin du makefile.

4.4 Un makefile générique

Après toutes ces explications, on aboutit à un makefile générique qui devrait vous servir pour des projets C de taille moyenne :

```
# Project specific variables
PROGRAM=
SOURCES=
SUBDIRS=
CC=
CFLAGS=
CPPFLAGS=
LDFLAGS=

# Generic part of the makefile
OBJECTS=$(patsubst %.c,%.o, $(SOURCES))

.PHONY: all clean

all: $(PROGRAM)

%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm --force $@.$$$$

$(PROGRAM):$(OBJECTS)

clean:
     for dir in $(SUBDIRS); do \
       $(MAKE) clean -C $$dir; \
     done;
     rm --force main *.o *~ \#*

include $(SOURCES:.c=.d)
```

4.5 Vers un makefile portable

Lorsqu'on veut que son projet compile sur plusieurs environnements, il faut se donner les moyens d'adapter les commandes du makefile en fonction des caractéristiques du système d'exploitation ou de la distribution visée. Par exemple, sur certains systèmes UNIX tels que SOLARIS, le compilateur C n'est pas GCC mais CC.

On a vu que l'alliance des variables de makefile et de la directive `include` permet de configurer la compilation. Il arrive que cela ne soit pas suffisant et que certains modules du projet ne soient carrément *pas compilables* sur le système. Cela se produit lorsque l'on utilise des extensions du compilateur GCC par exemple. Il faut alors désactiver la compilation de certains fichiers objets.

Make permet l'évaluation conditionnelle de makefile à l'aide des commandes `ifeq VAR VALUE` et `ifdef VARIABLE` qui permettent de tester la valeur d'une va-

riable ou bien, simplement, son existence. On peut donc rajouter les lignes suivantes dans un makefile.

```
# If we are working with gcc, we can compile some other features.
ifeq "CC" "gcc"
OBJECTS= ext_features.o $(OBJECTS)
endif

# If the Qt library is present, we can compile a GUI.
ifdef QT_IS_PRESENT
OBJECTS= gui.o $(OBJECTS)
endif
```

▷ Utilisez ces instructions de compilation conditionnelle pour compiler tantôt avec des appels récursifs de Make, tantôt avec des inclusions de makefiles.

5 Travail à remettre

Complétez l'archive avec les modifications apportées par ce sujet (toutes les lignes commencant par ▷).

6 Pour en savoir plus

La documentation de make s'obtient ainsi :

```
info make
```

Pour des projets de plus grande dimension et nécessitant une importante configurabilité, on utilise les AUTOTOOLS, c'est-à-dire autoconf et automake.