Make (Makefile)

make est un logiciel traditionnel d'UNIX. C'est un « moteur de production » : il sert à appeler des commandes créant des fichiers. À la différence d'un simple script shell, make exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes.

make fut à l'origine développé par le docteur Stuart Feldman, en 1977. Ce dernier travaillait alors pour Bell Labs.

Depuis, plusieurs dérivés ont été développés, les plus connus et utilisés sont ceux de BSD et celui de GNU, ce dernier étant généralement celui utilisé par défaut avec les systèmes Linux.

De nos jours, les fichiers Makefile sont de plus en plus rarement générés à la main par le développeur mais construit à partir d'outils automatiques tels qu'autoconf ou cmake qui facilitent la génération de Makefile complexes et spécifiquement adaptés à l'environnement dans lequel les actions de production sont censées se réaliser.

Introduction

- La fabrication logicielle se décompose en deux étapes :
  - compiler le(s) fichier(s) source en module(s) objet
  - lier le(s) module(s) objet et ses dépendances (librairies) pour fabriquer l'exécutable

Compilation séparée

- L'application est décomposée en plusieurs modules ou fichiers afin d'assurer une programmation modulaire plus compréhensible et une maintenance plus facile.

- Les besoins :
  - compilation séparée automatisée
  - optimisation du temps et des ressources : ne (re-)compiler que le nécessaire
  - mémoriser les règles de compilation de l'ensemble de l'application

L'OUTIL MAKE

- Make est un outil indispensable aux développeurs :

  - make assure la compilation séparée automatisée
  - make permet de ne recompiler que le code modifié
  - make utilise un fichier distinct contenant les règles de fabrication (Makefile)
  - make permet d'utiliser des commandes (ce qui permet d'assurer des tâches de nettoyage, d'installation, d'archivage, etc ...)
  - make utilise des variables, des directives, ...

Fichier Makefile

- Le fichier Makefile contient la description des opérations nécessaires pour générer une application :
  - les dépendances
  - les règles à appliquer lorsque les dépendances ne sont plus respectées
- Le fichier Makefile sera donc lu et exécuté par la commande make.
- Une règle est une suite d'instructions qui seront exécutées pour construire une cible, si la cible n'existe pas ou si des dépendances sont plus récentes. La syntaxe d'une règle est la suivante:

```
cible: dépendance(s)
<TAB>commande(s)
```

Exemple basique

La cible par défaut est la cible de la première règle trouvée par `make` (dans l'exemple c'est `morpion`, le plus souvent on utilisera la cible `all`)

morpion.cc

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Le jeu du morpion" << endl;
  return 0;
}
```

Fabrication

```
$ make

$ make morpion

$ make -f Makefile

$ make clean
```

```
morpion: morpion.o
    g++ morpion.o -o morpion

morpion.o: morpion.cc
    g++ -c -Wall morpion.cc

clean:
    rm -f *.o
```

Makefile

Les variables

- Déclaration : `NOM = VALEUR`
- Utilisation : `$(NOM)`
- Exemple :

```
CXX = g++ -c
LD = g++
CFLAGS = -Wall
TARGET=morpion # l'espace n'est pas obligatoire

all: $(TARGET)
$(TARGET): $(TARGET).o
    $(LD) $(TARGET).o -o $(TARGET)

$(TARGET).o: $(TARGET).cc
    $(CXX) $(CFLAGS) $(TARGET).cc
```

Les variables automatiques (1/2)

- Les variables automatiques sont des variables qui sont actualisées au moment de l'exécution de chaque règle, en fonction de la cible et des dépendances :
  - $@ : nom complet de la cible
  - $< : la première dépendance
  - $? : les dépendances plus récentes que la cible
  - $^ : toutes les dépendances
  - $* : correspond au nom de base (sans extension) de la cible courante
  - $$: le caractère $
  - ...

Les variables automatiques (2/2)

- Exemple :

```
CXX = g++ -c
LD = g++
CFLAGS = -Wall
TARGET = morpion

$(TARGET): $(TARGET).o
    $(LD) $^ -o $@

$(TARGET).o: $(TARGET).cc $(TARGET).h
    $(CXX) $(CFLAGS) $<
```

Les Règles génériques

- Exemple :

```
CXX = g++ -c
LD = g++
CFLAGS = -Wall
TARGET = morpion

$(TARGET): $(TARGET).o
    $(LD) $^ -o $@

$(TARGET).o: $(TARGET).h

%.o: %.cc                          # anciennement .cc.o:
    $(CXX) $(CFLAGS) -o $@ $<
```

Divers

- Génération de liste de fichiers objets :

```
SRC = morpion.cc testMorpion.c
OBJ = $(SRC:.cc=.o)
```

  - Ou de manière plus « robuste » :

```
OBJ = $(strip $(patsubst %.cc, %.o, $(wildcard *.cc)))
```

- Génération de la liste des fichiers sources :

```
SRC = $(wildcard *.cc)
OBJ = $(SRC:.cc=.o)
```

- Désactiver l'echo des lignes de commandes : ajouter le caractère @ devant la commande

La cible .PHONY (1/2)

- Dans le Makefile basique, la cible clean est la cible d'une règle ne présentant aucune dépendance.

- Problème : si un fichier porte le nom d'une cible, il serait alors forcément plus récent que ses dépendances et la règle ne serait alors jamais exécutée.

- Solution : il existe une cible particulière nommée .PHONY dont les dépendances seront systématiquement reconstruites.

La cible .PHONY : exemple (2/2)


```
CXX = g++ -c
LD = g++
CFLAGS = -Wall
TARGET = morpion

$(TARGET): $(TARGET).o
    $(LD) $^ -o $@

$(TARGET).o: $(TARGET).cc $(TARGET).h
    $(CXX) $(CFLAGS) $<

.PHONY: clean cleanall

clean:
    rm -f *.o

cleanall:
    rm -f *.o $(TARGET)
```

Les conventions (1/3)

- Noms d'exécutables et d'arguments
  - *Remarque : entre parenthèses les valeurs par défaut*
- AR: programme de maintenance d'archive (ar)
- CC: compilateur C (cc)
- CXX: compilateur C++ (c++)
- RM: commande pour effacer un fichier (rm)
- CFLAGS: paramètres à passer au compilateur C
- CXXFLAGS: paramètres à passer au compilateur C++
- LDFLAGS : paramètres à passer à l'éditeur de lien
- LIBS : les librairies préfixées par -l
- LIBSDIR : les chemins des répertoires contenant des librairies préfixées par -L
- INCLUDES : les chemins des répertoires contenant des fichiers headers préfixés par -I (-I.)
- WARN : les options de warning (-Wall)
- ...

Les conventions (2/3)

- Noms de répertoires de destination
  - *Remarque : entre parenthèses les valeurs par défaut*
- prefix: racine du répertoire d'installation (/usr/local)
- exec_prefix: racine pour les binaires ($(prefix))
- bindir: répertoire d'installation des binaires ($(exec_prefix)/bin)
- libdir: répertoire d'installation des librairies ($(exec_prefix)/lib)
- includedir: répertoire d'installation des en-têtes ($(prefix)/include)
- mandir: répertoire d'installation des fichiers de manuel ($(prefix)/man)
- srcdir: répertoire d'installation des fichiers source ($(prefix)/src)

- Exemple : permet de récupérer le répertoire courant, sinon utiliser le . (répertoire courant) :

```
prefix = $(shell sh -c pwd)
```

Les conventions (3/3)

- Noms de cibles : un utilisateur de make peut donner à ses cibles le nom qu'il désire. Mais pour des raisons de lisibilité, on donne toujours un nom standard à ses cibles selon leur comportement.
- Quelques exemples de cibles standards :
- all: compile tous les fichiers source pour créer l'exécutable principal
- install: exécute all, et copie l'exécutable, les librairies, les datas, et les fichiers en-tête s'il y en a dans les répertoires de destination
- uninstall: détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation (où se trouvent les fichiers source et le Makefile)
- clean: détruit tout les fichiers créés par all
- dist: crée un fichier tar de distribution
- dep: crée les dépendances automatiquement

➢ *Remarques : il est possible de regrouper l'ensemble des variables dans un fichier séparé (par exemple Makefile.cfg) et de l'inclure à partir du Makefile :*

```
include Makefile.cfg
```

Les directives

- On peut tester la définition d'une variable ou son contenu :

```
ifndef VERSION
VERSION = RELEASE
endif
...
ifeq (DEBUG, $(VERSION))
   CFLAGS = $(INCLUDES) $(WARN) $(DEBUG)
else
   CFLAGS = $(INCLUDES)
endif
...
# Il est ainsi 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

debug:
   @$(MAKE) VERSION=DEBUG
```

Le Makefile du TP1


```
CFLAGS=-Wall
CC=g++ -g -c $(CFLAGS)
LD=g++ -g -o

all: testRatio

testRatio: testRatio.o Ratio.o
   $(LD) $@ $^

testRatio.o: testRatio.cc Ratio.h
   $(CC) $<

Ratio.o: Ratio.cc Ratio.h
   $(CC) $<

clean:
   rm testRatio *.o *.*~
```

Bibliographie

- GNU Make : http://www.gnu.org/software/make/manual/html_node/
- Tutoriel Makefile : http://gl.developpez.com/tutoriel/outil/makefile/