Automatisation

- La compilation manuelle n'est pas possible sur un grand projet :
  - homogénéisation des options de compilation
  - gestion des dépendances (que doit-on recompiler?)
  - commandes et options spécifiques à une plate-forme.
  - erreurs manuelles
  - …
- On doit se munir d'outils pour automatiser cette étape.

Makefile

- make est un outil qui permet d'automatiser la création et la mise à jour de fichiers.
- make repose sur un fichier de description : Makefile
- Un Makefile est un ensemble de règles formatées :

  cible : source1 source2 source3 ...

  ```
  commande1
  commande2
  ... .
  ```

- Si « cible » n'existe pas où est moins récente que l'une des sources, alors les commandes sont exécutées.

Makefile : exemple

```
image_thumbail.jpg : image.jpg
	convert -size 80x80 image.jpg image_thumbnail.jpg
image.jpg : image.png
	convert image.png image.jpg
image.png : image.svg
	inkscape -z -e image.png image.svg
```

- make est récursif : si un fichier source n'existe pas, il cherche une règle pour le créer.
- initialement, make cherche à créer une seul cible : la première du Makefile ou celle(s) indiquée(s) sur la ligne d'appel : make image.png
- Si une source est manquante et qu'il n'y a pas de règle pour la créer : erreur
- Si une des commandes renvoie une valeur différente de 0, make s'arrête (tips : `commande || true`)

Makefile : variable

- make supporte la déclaration de variable :

  `NOM=VALEUR`

- valeur peut comporter des espaces

  `NOM-=VALEUR`

- Si NOM est dans l'environnement, alors c'est la valeur de l'environnement qui est utilisée, sinon c'est VALEUR.
- $(NOM) est remplacé par VALEUR.

Makefile : variable, exemple

```
CONVERT=convert
THUMB_SIZE-=80x80
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
image.jpg : image.png
	$(CONVERT) image.png image.jpg
image.png : image.svg
	inkscape -z -e image.png image.svg
```

$> THUMB_SIZE=60x60 make

Makefile : spéciales

- Quelques variables spéciales pour l'écriture des commandes :

```
cible : source1 source2
```

  - $@ : cible
  - $< : source1
  - $^ : source1 source2

- Ainsi :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
```

- Devient :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) $< $@
```

Makefile : règles génériques

- Il est possible d'écrire des règles génériques :

```
%.jpg : %.png
	convert $< $@
```

- Permet de convertir tout fichier png en jpg à l'aide du programme *convert*

```
%_thumb.jpg : %.jpg
	convert -size 80x80 $< $@
```

- Il est enfin possible de séparer la liste des dépendances et la règle de création.

Makefile et compilation

```
CC=gcc
CFLAGS-=-Wall

prog : example.o hash.o
	$(CC) -o $@ $^
exemple.o : exemple.c hash.h
hash.o : hash.c hash.h
%.o :
	$(CC) $(CFLAGS) $< -o $@
```

- La dernière règle explique comment générer un fichier .o
- les deux règles au dessus donnent les dépendances

Dépendances

- C’est au développeur d’indiquer les dépendances.
- Ainsi, si un fichier header est modifié, tout fichier source qui inclut directement ou indirectement ce header doit être recompilé
- On peut demander à gcc d’analyser les fichiers et produire les règles de dépendances:

```
$> gcc -MM hash.c exemple.c
hash.o: hash.c hash.h
exemple.o: exemple.c hash.h
```

- Le résultat peut être inclus dans le Makefile avec la fonction *include*

include et gcc -MM

```
CC=gcc
CFLAGS=-Wall

prog: hash.o exemple.o
	$(CC) $^ -o $@

include dep
dep: hash.c exemple.c hash.h
	gcc -MM $^ > dep
```

```
pg106$ make
make: 'prog' is up to date.
pg106$ touch exemple.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.h
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc hash.o exemple.o -o prog
pg106$
```

Makefile : PHONY

- On peut vouloir écrire des règles qui ne génèrent pas de fichier :

```
all

install

clean

distclean
```

- On indique cela à make :

```
.PHONY=all install clean distclean
```

Makefile, automake...

- Makefile n'est pas spécifique à la compilation de programme
- Il ne gère pas nativement la dépendance entre les fichiers sources
- La prise en compte de l'environnement (compilateur, options, bibliothèques…) peut ce faire :
  - par l'édition des variables du Makefile (options de compilations, répertoire d'installation...)
  - par l’écriture de plusieurs Makefile (un par système)
  - par l'utilisation d'un générateur :
    - automake / autoconf
    - cmake

Exercices

- Reprendre votre script de compilation.
- Créer un *Makefile* par étapes successives :
  - Commencer par la compilation des fichiers *example.o, hello.o*
  - Faire bien attention aux dépendances !!
  - Ajouter une règle pour créer le binaire *example*
  - Ajouter une règle pour la création de la bibliothèque statique et l'exécutable utilisant cette bibliothèque.
  - Ajouter une règle pour la compilation de bibliothèque dynamique (attention au binaire) et l'exécutable utilisant cette bibliothèque.
- Ajouter une règle *all* qui compile tous les exécutables et bibliothèques
- Ajouter une règle *clean* qui supprime tout les fichiers construits

Exercices

- Faire une deuxième version de votre *Makefile* (*MakefileV2*?)
- Comment demander à *make* d'utiliser le fichier *MakefileV2* ?
- Écrire une règle générique pour la compilation des objets à partir des sources
- Déclarer des variables CC, CFLAGS et LDFLAGS qui prendront comme valeurs par défaut gcc , -Wall et rien. Si ces variables sont dans l'environnement, alors il faut les laisser ainsi (*?*=).
- Mettre à jour votre fichier en fonction de ces variables
- Tester la commande *gcc -MM exemple.c hello.c*
- Créer une règle *depends* qui crée un fichier contenant le résultat de cette commande et ajouter l'inclusion de ce même fichier dans votre MakefileV2