Compilation `C` sous *Unix*

Épisode 2: makefile

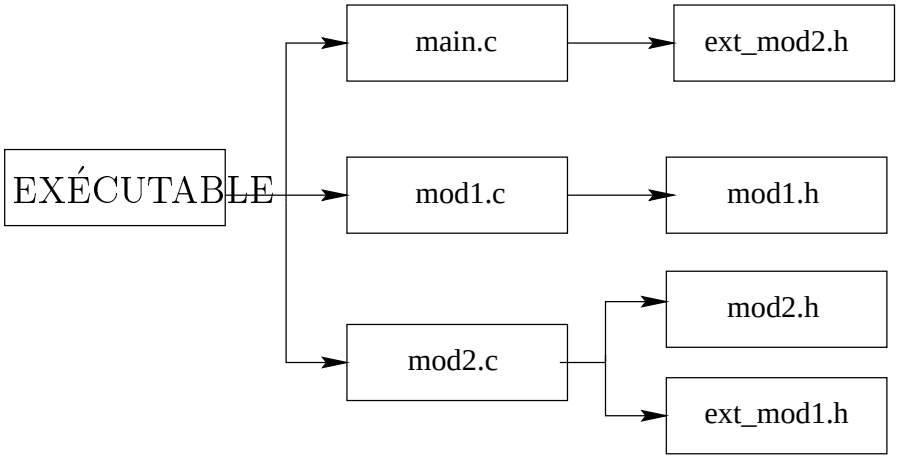
`Make` et compilation séparée

La conception modulaire d'un programme présente des inconvénients pratiques. Plus un projet est important, plus il est difficile de gérer les problèmes de dépendance entre modules et de compilation de l'ensemble.

La commande `make` permet de résoudre les problèmes de la gestion d'un programme modulaire en apportant:

- une gestion simple de l'ensemble des modules.
- une gestion automatique des dépendances et des recompilations. En particulier, on ne recompile que ce qui est nécessaire (gain de vitesse dans le processus de développement).
- une automatisation des tâches (construction de bibliothèques, nettoyage, installation, ...)

Exemple de relations de dépendance



La commande `make` fonctionne à partir d'un fichier `makefile` écrit par le programmeur qui décrit:

- Les relations de dépendance entre les différents modules ou tâches.
- Les règles de construction et d'assemblage des modules.
- Les tâches à automatiser.
- Le paramétrage des tâches et des règles.

Règles de dépendance

objet : fichier préexistant (`.c`) ou obtenu par construction (`.o` par compilation du `.c`).

label : symbole défini dans le `makefile` associé à une tâche.

Écriture de la règle de dépendance:

A: B_1 B_2
<TAB>*tâches*

où A, B_1 et B_2 sont des *objets* ou des *labels*. Il ne doit y avoir qu' **une seule règle** associée à A (appelée souvent la cible).

Relation de dépendance:

A dépend de B_1 et B_2.

Résolution de la règle:

Résoudre les B_i pour tout i.
Résoudre A

Résolution d'un B_i

Rechercher la règle de dépendance du B_i et si elle existe l'appliquer.

Résolution de A

SI A est un *objet*
ALORS s'il n'existe pas ou s'il est plus ancien que les *objets* B_i dont il dépend, exécuter la *tâche*.

SI A est une *label*
ALORS exécuter la *tâche*.

Exemple

```
main.o: ext_mod2.h
    gcc -c main.c
mod1.o: mod1.h
    gcc -c mod1.c
mod2.o: ext_mod1.h mod2.h
    gcc -c mod2.c
main: main.o mod1.o mod2.o
    gcc -o main main.o mod1.o mod2.o
```

Règles de dépendance conditionnelle

Écriture de la règle de dépendance conditionnelle :

A:: B_1
<TAB>*tâches1*
A:: B_2
<TAB>*tâches2*

où A est un *objet* ou un *label*, B_1 et B_2 sont des *objets*.

Il peut y avoir autant de règles que nécessaire.

Relations de dépendance:

A dépend de B_1.
A dépend de B_2.

Résolution de la règle:

SI (A n'existe pas) ou (A est plus ancien que B_1)
ALORS exécuter la *tâche1*

SI (A n'existe pas) ou (A est plus ancien que B_2)
ALORS exécuter la *tâche2*

Dépendance: cas particulier des bibliothèques:
Si A est une bibliothèque, alors la dépendance des B_i doit s'écrire $A(B_i)$ pour indiquer que c'est la date de B_i dans la bibliothèque A qui doit être testée.

Exemple

```
mylib:: mylib(mod1.o)
    ar -r mylib.a mod1.o
mylib:: mylib(mod2.o)
    ar -r mylib.a mod2.o
```

Règles implicites

ou règles d'inférence

But: il paraît fastidieux d'avoir à entrer toujours la règle de construction d'un `.o` à partir d'un `.c` alors que celle-ci est généralement toujours la même. La commande `make` permet de définir pour des couples de suffixesdes règles par défaut de passage de l'un à l'autre.

Déclaration de nouveaux suffixes : la commande déclare les suffixes **.deb**, **.fin** et **.xxx**.

```
.SUFFIXES: .deb .fin .xxx
```

Déclaration d'une règle implicite : cette règle définit une règle de construction par défaut permettant de construire un fichier *nom*.**fin** à partir du fichier *nom*.**deb**.

```
.deb.fin:
<TAB> tâche
```

Conséquence: il suffit juste alors de définir les dépendances, car les *tâches* de construction deviennent implicites.

Macros prédéfinies pour une règle implicite:

$<	*nom*.**deb**
$*	*nom*

Exemple

```
.SUFFIXES: .o .c
.c.o:
    gcc -c $<
main.o: ext_mod2.h
mod1.o: mod1.h
mod2.o: ext_mod1.h mod2.h
main: main.o mod1.o mod2.o
    gcc -o main main.o mod1.o mod2.o
```

- Ici, toutes les tâches de construction des `.o` sont **implicites**.
- Une écriture alternative de `gcc -c $<` est `gcc -c $*.c`

Définitions de macros

macro : variable utilisée pour rendre le `makefile` facilement modulable.

Définition d'une macro:

NOM=*valeur*

Utilisation d'une macro:

`$(`*NOM*`)`

Exemple:

```
CC=gcc
DIR=~/bin
LIBS=-lc -lm
OBJS=main.o mod1.o mod2.o
main: $(OBJS)
    $(CC) -o $(DIR)/main $(OBJS) $(LIBS)
```

Macros, règles et suffixes prédéfinis

Les macros, règles et suffixes les plus usuels sont la plupart du temps déjà prédéfinis pour `make`. On peut consulter les définitions en utilisant la commande: `make -p`

Exemples d'utilisation des règles prédéfinies:

```
$ make toto
cc     toto.c   -o toto
$ cat makefile
main.o: ext_mod2.h
mod1.o: mod1.h
mod2.o: ext_mod1.h mod2.h
main: main.o mod1.o mod2.o
$ make main
cc    -c main.c -o main.o
cc    -c mod1.c -o mod1.o
cc    -c mod2.c -o mod2.o
cc   main.o mod1.o mod2.o   -o main
```

Aspects avancés

Suffixes, règles et macros

Macros prédéfinie dans les *tâches* d'une règle de dépendance:

Pour les fichiers: A: B_1 B_2	`$@`	A
	`$*`	A sans extension
	`$?`	les B_i plus récents que A
Pour les bibliothèques: A: $A(B_1)$ $A(B_2)$	`$@`	A
	`$%`	les B_i plus récents que A (aussi pour les inférences)

Exemple:

```
MODULES= mod1.o mod2.o
main.o: $(MODULES)
    gcc -c $@
    gcc -o $* $@ $(MODULES)
    cp $? ./Backup
```

Substitutions de suffixes dans une macro : on substitue dans la macro `$`*NOM* les extensions **.deb** par les extensions **.fin** avec la macro:

`$(`*NOM*`:`**.deb=.fin**`)`

Exemple:

```
SRCS= main.c mod1.c mod2.c
OBJS= $(SRCS:.c=.o)
main: $(OBJS)
    gcc -o $@ $(OBJS)
```

Désactivation d'une règle implicite préexistante

.deb.fin: ;

Commande `include` *fichier*

Elle permet d'inclure un *fichier* dans le `makefile`. Ce *fichier* est alors considéré comme partie intégrante du `makefile`. Toute dépendance, règle ou macro du *fichier* s'applique lors du `make`.

Notes sur les *tâches*

- Une *tâche* est une instruction ou un ensemble d'instructions `sh` utilisant éventuellement les macros prédéfinies ou définies dans le `makefile`.
- Par défaut, toutes les *tâches* exécutées sont affichées sur la sortie standard.
- Par défaut, si une *tâche* produit une erreur, le `make` s'arrête immédiatement, sans exécuter les *tâches* suivantes.
- Les variables d'environnement et les aliases peuvent être utilisés dans un `makefile`. Ce n'est pas **du tout** conseillé car le `makefile` ne devient plus portable.
- Pour les commandes standards comme `rm`, utiliser plutôt le chemin complet `/bin/rm` afin d'éviter d'utiliser une version aliasée par l'utilisateur.

Caractères spéciaux de début de ligne:

Les caractères suivants permettent de modifier localement le comportement du `make` lors de l'exécution d'une tâche.

- `-` en cas d'erreur, poursuite du `make`.
- `+` ligne toujours exécutée même en mode verbose.
- `@` pas d'affichage de la tâche lors de l'exécution.

Exemple:

```
$ echo $PWD
/home/pascal/Universite/StageUnix2/Exemples
$ cat makefile
cat makefile
test:
        @echo $(PWD)
        cd ...
        ls
$ make test
/home/pascal/Universite/StageUnix2/Exemples
cd ...
/bin/sh: cd: ...: No such file or directory
make: *** [test] Error 1
$
```

Gestion automatique des dépendances

L'utilitaire `makedepend` permet de générer automatiquement les dépendances d'un ensemble de fichiers. Par défaut, les règles de dépendance sont insérées dans le fichier `makefile` situé dans le répertoire courant.

Exemple:

```
$ cat makefile
main: main.o mod1.o mod2.o
$ makedepend mod*.c main.c
$ cat makefile
main: main.o mod1.o mod2.o

mod1.o: mod1.h
mod2.o: ext_mod1.h mod2.h
main.o: ext_mod2.h
$
```

Options de `makedepend`:

`-D`*nom*`=`*valeur*	définition de constantes pour le précompilateur
`-D`*nom*	
`-a`	ajoute les dépendances (au lieu de les remplacer)
`-f`*fichier*	sortie dans *fichier*.
	- rédirige sur la sortie standard.
`--` *options* `--`	*options* spécifiques du compilateur

Note:

L'option `-M` de `gcc` permet d'afficher les dépendances détectées pour un fichier `.c`.

```
$ gcc -M mod2.c
mod2.o: mod2.c ext_mod1.h mod2.h
```

Notes générale sur les `makefile`

- Sans options, le `make` exécute la première règle de dépendance du fichier `makefile`.
- Comme dans un shellscript, le caractère pour insérer des lignes de commentaire est `#`.
- Le caractère \ permet de couper les lignes en les poursuivant à la ligne suivante.

Invocation de `make`

Options en ligne

nom	exécute la règle dont la cible (i.e. le A) est *nom*.
`-n`	affiche les commandes sans les exécuter.
`-d`	mode débug.
`-i`	pas d'arrêt sur erreur.
`-k`	en cas d'erreur, abandon de la branche courante, mais poursuite du reste.
`-s`	mode sans affichage des commandes.
`-p`	affiche l'ensemble des macros et règles prédéfinies.

Options par défaut

Il existe des commandes spéciales qui, insérées dans le code du `makefile`, provoquent des comportements par "défaut".

`.IGNORE`	pas d'arrêt sur erreur.
`.SILENT`	mode sans affichage des commandes.

On peut également définir la macro `MAKEFLAGS` dans le corps du makefile. On a alors les équivalences suivantes:

makefile	⇔	*makefile*	⇔	*shell*
`.IGNORE`		`MAKEFLAGS = i`		`make -i`

Pour relancer une compilation complète: 2 options

- Effacer tous les `.o`.
- Utiliser la commande *shell* `touch` sur tous les `.c`. Elle actualise la date d'un fichier (i.e. comme s'il venait d'être modifié).

Compilation dans des répertoires séparés

Voici quelques outils utiles à cet effet:

- La macro `VPATH` permet de définir les répertoires dans lesquels `make` va chercher pour compiler.
- Il existe des modificateurs sur toutes les macros (y compris celles issues des règles). Dans le cas où la macro A est sous la forme *répertoire*/*nom*:

`$(@F)`	*nom*
`$(@D)`	*répertoire*

- Les répertoires des includes sont fixés avec l'option `-I`*includedir* de `gcc`. Elle permet d'ajouter aux chemins de recherche utilisateur (i.e. `#include "..."`) le répertoire *includedir*.
- Les répertoires des bibliothèques utilisateurs sont fixés avec l'option `-L`*libdir* de `gcc`.
- Avoir un `makefile` par répertoire est souvent une bonne idée.

Exemples d'utilisation des règles prédéfinies:

```
$ find .
./makefile
./main.c
./module1
./module1/mod1.c
./module1/mod1.h
./module1/ext_mod1.h
./module2
./module2/mod2.c
./module2/mod2.h
./module2/ext_mod2.h
$ cat makefile
VPATH = ./module1 ./module2
CFLAGS = -I./module1 -I./module2
main: main.o mod1.o mod2.o
$ make main
cc -I./module1 -I./module2   -c main.c -o main.o
cc -I./module1 -I./module2   -c ./module1/mod1.c -o mod1.o
cc -I./module1 -I./module2   -c ./module2/mod2.c -o mod2.o
cc   main.o mod1.o mod2.o   -o main
```