L'outil make - Ne *re*faire que le nécessaire

Programmation Système — R3.05

C. Raïevsky
Avec la courtoisie de Damien Genthial

IUT A Valence
Université Grenoble Alpes

Département Informatique
BUT Informatiques 2ème année

Plan du chapitre

Automatiser mais pas uniquement

Make sert à automatiser des tâches

Quelle différence avec un script bash ?

Automatiser mais pas uniquement

Make surveille des dépendances et ne fait que le nécessaire

En utilisant des règles de la forme :

```
Cible : dépendances
	Commandes pour générer la cible
```

Exemple de compilation

```
executable : source.c source.h
	gcc source.c -o executable
```

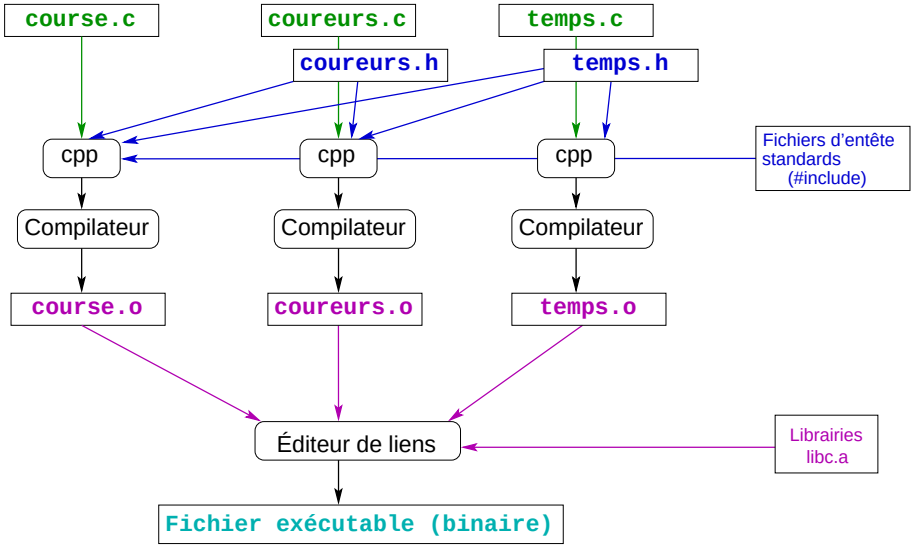
Exemple

Traiter des notes

- Mettre à jour un fichier de sauvegarde
- Générer un fichier trié :
  - par ordre alphabétique,
  - par note.
- Établir des dépendances entre cibles
- Transformer un fichier `txt` en `csv`
- Faire le ménage
- Définir et utiliser des variables

Faciliter la compilation

Cas typique d'utilisation



À chaque modification des sources

Compilation des modules et du programme principal

```
gcc -c -Wall coureurs.c                    → coureurs.o
gcc -c -Wall temps.c                       → temps.o
gcc -c -Wall course.c                      → course.o
```

Création de l'exécutable (édition de liens)

```
gcc -o course course.o coureurs.o temps.o
                                           → course
```

Automatiser avec un script

- soit il doit tout refaire à chaque fois ;
- soit il faut prévoir un script de compilation, un d'installation,...

À chaque modification des sources

Compilation des modules et du programme principal

```
gcc -c -Wall coureurs.c                    → coureurs.o
gcc -c -Wall temps.c                          → temps.o
gcc -c -Wall course.c                         → course.o
```

Création de l'exécutable (édition de liens)

```
gcc -o course course.o coureurs.o temps.o
                                              → course
```

Automatiser avec un script

- soit il doit tout refaire à chaque fois ;
- soit il faut prévoir un script de compilation, un d'installation,...

À chaque modification des sources

Compilation des modules et du programme principal

```
gcc -c -Wall coureurs.c                    → coureurs.o
gcc -c -Wall temps.c                          → temps.o
gcc -c -Wall course.c                         → course.o
```

Création de l'exécutable (édition de liens)

```
gcc -o course course.o coureurs.o temps.o
                                              → course
```

Automatiser avec un script

- soit il doit tout refaire à chaque fois ;
- soit il faut prévoir un script de compilation, un d'installation,...

Utilité de make

Exécuter des commandes

```
gcc -c -Wall course.c
cp course $HOME/bin
rm -f *.o
```

Uniquement si c'est nécessaire

- En fonction de l'existence d'un fichier ou de sa date de modification
- Utilisation des *dépendances* entre fichiers

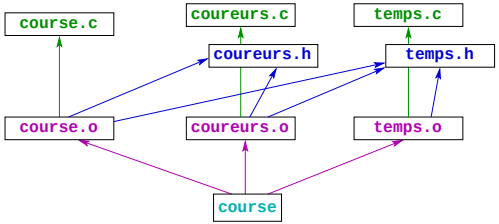
Utilité de make

Exécuter des commandes

```
gcc -c -Wall course.c
cp course $HOME/bin
rm -f *.o
```

Uniquement si c'est nécessaire

- En fonction de l'existence d'un fichier ou de sa date de modification
- Utilisation des *dépendances* entre fichiers



Le fichier Makefile

Une liste de règles de la forme :

```
Cible : dépendances
	Commandes pour générer la cible
```

Lignes de dépendances

Commencent à gauche (colonne 1)

Format

```
Liste de cibles: liste des cibles requises
```

- Les cibles sont en général des noms de fichiers
- Mais une cible peut ne pas correspondre à un fichier

Exemple

```
temps.o: temps.c temps.h
```

Se lit : `temps.o` dépend de `temps.c` et de `temps.h`
Donc si `temps.c` et/ou `temps.h` sont modifiés, `temps.o` n'est plus à jour
$\Longrightarrow$ nécessité de reconstruire la *cible* `temps.o`

Commandes

Suivent une ligne de dépendances

Ligne suivante, après une *tabulation* (pas des espaces !)

```
temps.o: temps.c temps.h
Tabulation cc -c -Wall -g temps.c
```

Sur la même ligne après un ';'

```
install: course ; cp course $(HOME)/bin
```

Permettent de reconstruire la cible

Mais pas forcément

```
^^Ipropre:
	rm -f *.o *~ course
```

Commandes

Suivent une ligne de dépendances

Ligne suivante, après une *tabulation* (pas des espaces !)

```
temps.o: temps.c temps.h
Tabulation cc -c -Wall -g temps.c
```

Sur la même ligne après un ';'

```
install: course ; cp course $(HOME)/bin
```

Permettent de reconstruire la cible

Mais pas forcément

```
^^Ipropre:
	rm -f *.o *~ course
```

Commandes

Suivent une ligne de dépendances

Ligne suivante, après une *tabulation* (pas des espaces !)

```
temps.o: temps.c temps.h
Tabulation cc -c -Wall -g temps.c
```

Sur la même ligne après un ';'

```
install: course ; cp course $(HOME)/bin
```

Permettent de reconstruire la cible

Mais pas forcément

```
^^Ipropre:
	rm -f *.o *~ course
```

Commandes

Suivent une ligne de dépendances

Ligne suivante, après une *tabulation* (pas des espaces !)

```
temps.o: temps.c temps.h
Tabulation cc -c -Wall -g temps.c
```

Sur la même ligne après un ';'

```
install: course ; cp course $(HOME)/bin
```

Permettent de reconstruire la cible

Mais pas forcément

```
^^Ipropre:
	rm -f *.o *~ course
```

Macros

Définition

- Ce sont des variables (comme celles du shell)
- Intérêt : paramétrage, une seule définition, plusieurs utilisations

```
CC=cc -c -Wall
```

Utilisation

- Avec un $ comme dans le shell
- Beaucoup de macros prédéfinies
  ⇒ parenthèses obligatoires

```
temps.o: temps.c temps.h
   $(CC) temps.c
```

Les variables du shell peuvent être utilisées comme des macros

```
install: course ; cp course $(HOME)/bin
```

Macros

Définition

- Ce sont des variables (comme celles du shell)
- Intérêt : paramétrage, une seule définition, plusieurs utilisations

```
CC=cc -c -Wall
```

Utilisation

- Avec un $ comme dans le shell
- Beaucoup de macros prédéfinies
  ⇒ parenthèses obligatoires

```
temps.o: temps.c temps.h
   $(CC) temps.c
```

Les variables du shell peuvent être utilisées comme des macros

```
install: course ; cp course $(HOME)/bin
```

Macros

Définition

- Ce sont des variables (comme celles du shell)
- Intérêt : paramétrage, une seule définition, plusieurs utilisations

```
CC=cc -c -Wall
```

Utilisation

- Avec un $ comme dans le shell
- Beaucoup de macros prédéfinies
  ⇒ parenthèses obligatoires

```
temps.o: temps.c temps.h
   $(CC) temps.c
```

Les variables du shell peuvent être utilisées comme des macros

```
install: course ; cp course $(HOME)/bin
```

Inclusion de fichiers, règles

Inclusion de fichiers

```
include ../makefile.inc
```

- Permettent de découper un gros Makefile en plusieurs morceaux
- Dans les gros projets
  - Importer des définitions globales
  - Avoir un `Makefile` dans chaque sous-répertoire

Fonctionnement

Reconstruction d'une cible

```
make course
```

- Évaluation de toutes les règles ayant cette cible en partie gauche (à gauche des ':')

Évaluation d'une règle

- Reconstruire toutes les cibles prérequises (si nécessaire)
- Si une des cibles prérequises est plus récente que la cible courante, exécuter les commandes de la règle

Cas particuliers

- Si une cible n'est pas un fichier, ou que le fichier n'existe pas, elle est toujours considérée comme n'étant pas à jour

Fonctionnement

Reconstruction d'une cible

```
make course
```

- Évaluation de toutes les règles ayant cette cible en partie gauche (à gauche des ':')

Évaluation d'une règle

- Reconstruire toutes les cibles prérequises (si nécessaire)
- Si une des cibles prérequises est plus récente que la cible courante, exécuter les commandes de la règle

Cas particuliers

- Si une cible n'est pas un fichier, ou que le fichier n'existe pas, elle est toujours considérée comme n'étant pas à jour

Fonctionnement

Reconstruction d'une cible

```
make course
```

- Évaluation de toutes les règles ayant cette cible en partie gauche (à gauche des ':')

Évaluation d'une règle

- Reconstruire toutes les cibles prérequises (si nécessaire)
- Si une des cibles prérequises est plus récente que la cible courante, exécuter les commandes de la règle

Cas particuliers

- Si une cible n'est pas un fichier, ou que le fichier n'existe pas, elle est toujours considérée comme n'étant pas à jour

Lancement de make

```
make [-f fichier_règles] [options]
     [macro=valeur] [noms_de_cible ...]
```

Exemples :

`make`	le fichier par défaut est `makefile`, `Makefile`, `s.makefile` et `s.Makefile`
`make -f mesRegles`	`mesRegles` remplace `makefile`
`make prog`	reconstruit la cible `prog`
`make CC="gcc -ansi" prog`	
`make -n prog`	Affiche les commandes sans les exécuter (utile pour la mise au point).

Règles à motif (*Pattern rules*)

`%.o: %.c` ⇒ `<qqc>.o` dépend de `<qqc>.c`

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

Règle pour compiler les `.o` si les `.c` ou les `.h` correspondants sont modifiés

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

⇒ Tous les `.o` dépendent des `.c` des `.h` correspondants ainsi que de `globals.h`

Règles à motif (*Pattern rules*)

`%.o: %.c` ⇒ <qqc>`.o` dépend de <qqc>`.c`

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

Règle pour compiler les `.o` si les `.c` ou les `.h` correspondants sont modifiés

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

⇒ Tous les `.o` dépendent des `.c` des `.h` correspondants ainsi que de `globals.h`

Macros prédéfinies

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

$@	nom complet de la cible courante
$?	liste des pré-requis à mettre à jour dans la règle courante
$<	fichier source dans une règle
$*	nom de la cible (sans l'extension)

make coureurs.o

⇒ $@ = coureurs.o

⇒ $* = coureurs

⇒ $< = coureurs.c

⇒ $? = coureurs.h temps.h

Macros prédéfinies

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

$@	nom complet de la cible courante
$?	liste des pré-requis à mettre à jour dans la règle courante
$<	fichier source dans une règle
$*	nom de la cible (sans l'extension)

make coureurs.o

⇒ $@ = coureurs.o

⇒ $* = coureurs

⇒ $< = coureurs.c

⇒ $? = coureurs.h temps.h