

Compilation avec Make

Laurent Réveillère

Enseirb-Matmeca / IPB
Département Télécommunications

`Laurent.reveillere@ipb.fr`
`http://http://reveille.vvv.enseirb-matmeca.fr/`

Techniques de programmation avancée

Objectifs

- ◆ **Automatiser** la reconstruction d'un exécutable composé d'un ou plusieurs:
 - Modules objets
 - Librairies
 - Fichiers entêtes ...
- ◆ **Avantages**
 - Assure la compilation séparer des différentes ressources
 - Utilise des macro-commandes et des variables
 - Permet ne recompiler que le code nécessaire
 - Permet d'utiliser des commandes shell arbitraire
 - » Scripts d'installations
 - » Scripts de désinstallation
 - » Scripts de nettoyage
 - » Etc.

Quelques Définitions

- ◆ **Cible**
 - Exécutable à (re)construire
 - Commande à exécuter (installation, nettoyage, ...)
- ◆ **Dépendances**
 - Élément(s) dont dépend la cible
 - Exemple en C
 - » Cible dépendant d'un fichier source .c
 - » Cible construite que si le fichier source est plus récent que le fichier de la cible
- ◆ **Commande**
 - Commande(s) permettant de reconstruire la cible
- ◆ **Fichier makefile**
 - Fichier contenant la définition des dépendances et des règles de reconstruction des cibles

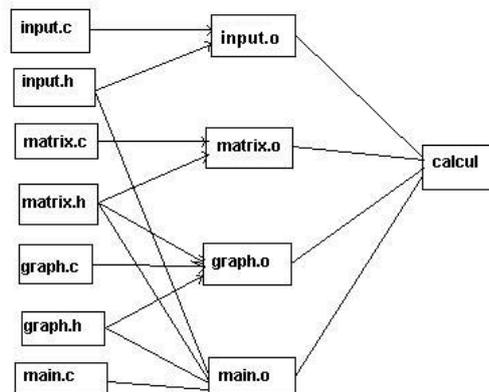
Techniques de programmation avancée

3

Exemple

Le programme « calcul » se décompose en 4 fichiers

- ◆ **input.c**
 - fonctions de lecture des fichiers de données
- ◆ **matrix.c**
 - fonctions de calcul matriciel
- ◆ **graph.c**
 - fonction d'affichage graphique
- ◆ **main.c**
 - programme principal
- ◆ **input.h, matrix.h, graph.h**
 - fichiers d'entêtes



Techniques de programmation avancée

4

Premier Makefile

- ◆ Permet de définir les cibles, les dépendances et les commandes pour reconstruire les cibles

```
calcul.o: main.o input.o matrix.o graph.o
    gcc -o calcul main.o input.o matrix.o graph.o
main.o: main.c input.h matrix.h graph.h
    gcc -c main.c
input.o: input.c input.h
    gcc -c input .c
matrix.o: matrix.c matrix.h
    gcc -c matrix.c
graph.o: graph.c graph.h matrix.h
    gcc -c graph.c
```

Règles de reconstruction d'une cible

- ◆ Forme générale

```
CIBLE (s) :DEPENDANCES
<tab>COMMANDE (s)
```

- ◆ Contraintes syntaxiques

- Cibles et dépendances sont sur une même ligne
- Le caractère « \ » peut forcer make à ignorer le retour à la ligne
- Commandes commencent toujours par une tabulation
- Commandes sur plusieurs lignes
 - » chaque ligne est séparée par le caractère « \ »
 - » Sauf la dernière ligne

Commandes

- ◆ Exécution d'une commande
 - ❑ Texte de la commande affichée sur la sortie standard
 - ❑ Préfixer la commande par le caractère « @ » pour ne pas l'afficher
 - ❑ Chaque ligne de commande est exécutée par un interpréteur différent du shell (sous Unix /bin/sh)
 - ❑ Mettre plusieurs commandes sur une même ligne ou séparées par « \ » pour utiliser le même interpréteur
- ◆ Comportement en cas d'erreur
 - ❑ L'exécution d'une règle et tout le processus sont abandonnés
 - ❑ Indiquer « - » en début de commande pour indiquer à make qu'il ignore l'échec de cette commande
 - ❑ Interruption d'un make en cours d'exécution d'une règle
 - » La cible est détruite par défaut

Techniques de programmation avancée

7

Exemples

```
dir/prog: a.o b.o
cd dir
gcc -o prog ../a.o ../b.o
```

Incorrect

```
dir/prog: a.o b.o
cd dir ;\
gcc -o prog ../a.o ../b.o
```

```
clean:
-rm *.o
@echo "done"
```

Techniques de programmation avancée

8

Évaluation d'un fichier Makefile

- ◆ Commande make sans argument
 - ❑ La première règle rencontrée est évaluée
- ◆ Commande make avec argument
 - ❑ Le nom de la règle passée en argument est évaluée
- ◆ Évaluation d'une règle
 1. Analyse des dépendances
 2. Si une dépendance est la cible d'une autre règle du Makefile, cette règle est à son tour évaluée.
 3. Lorsque l'ensemble des dépendances est analysé et si la cible ne correspond pas à un fichier existant ou si un fichier dépendance est plus récent que la règle, les différentes commandes sont exécutées

Cas d'étude

```
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"

void
Hello(void) {
    printf("%s\n", HELLO_MSG);
}
```

hello.c

```
#ifndef __HELLO_H__
#define __HELLO_H__

#define HELLO_MSG "Hello World"
extern void Hello(void);

#endif /* __HELLO_H__ */
```

hello.h

```
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"

int
main(int argc, char **argv) {
    Hello();
    return EXIT_SUCCESS;
}
```

main.c

Compilation avec make

\$ **emacs** Makefile

```
hello: hello.o main.o
    gcc -o hello hello.o main.o
hello.o: hello.c hello.h
    gcc -o hello.o -c hello.c -Wall -ansi
main.o: main.c hello.h
    gcc -o main.o -c main.c -Wall -ansi
```

\$ **make**

Que se passe t-il?

1. La première règle du Makefile est la règle `hello`
2. Évaluation de la règle `hello`
3. Analyse des dépendances
 1. Évaluation des cibles `hello.o` et `main.o`
 2. Analyses des dépendances pour la cible `hello.o`
 3. Analyses des dépendances pour la cible `main.o`
 4. Exécution des commandes si nécessaires
4. Exécution de la commande si nécessaire

Exercice

- Regarder les fichiers générés par `make`

Limitations

- ◆ Difficile de gérer plusieurs exécutable pour des plateformes différentes
- ◆ Tous les fichiers intermédiaires sont conservés
 - Pollution du répertoire de travail
- ◆ Impossible de forcer la régénération complète du projet
 - Suppression manuel des fichiers temporaires obligatoire

Cibles standard

- ◆ `all`
 - Première règle explicite pour définir ce qui est fait par défaut
- ◆ `install`
 - Installation du programme construit
 - Création de la structure de répertoire, copies des fichiers, ...
- ◆ `uninstall`
 - Défait ce que `install` a fait
 - Ne supprime pas les fichiers temporaires de la compilation
- ◆ `clean`
 - Supprime du répertoire courant tous les fichiers construits par `make all`
- ◆ `mrproper`
 - Remet les répertoires sources dans leur état initial

Application

```
all: hello

hello: hello.o main.o
    gcc -o hello hello.o main.o
hello.o: hello.c hello.h
    gcc -o hello.o -c hello.c -Wall -ansi
main.o: main.c hello.h
    gcc -o main.o -c main.c -Wall -ansi

clean:
    rm -f *.o
mrproper: clean
    rm -f *~ hello
```

Notion de variables

- ◆ Motivations
 - Simplifie l'évolution des règles
 - Factorise la définition des outils utilisés (compilateur, ...)
- ◆ Définition
 - NOM_VARIABLE = VALEUR
 - Contrainte sur le nom
 - » séquence de caractères sauf :, # = <tab> espace
- ◆ Utilisation
 - \$(NOM_VARIABLE)
- ◆ Exemple

```
CC = gcc
foo.o: foo.c
    $(CC) -c foo.c -o foo.o
```

Noms de variables standards

- ◆ AR
 - Programme de maintenance d'archive
- ◆ CC
 - Compilateur C
- ◆ CXX
 - Compilateur C++
- ◆ RM
 - Commande pour effacer un fichier
- ◆ ARFLAGS
 - Paramètres à passer au programme de maintenance d'archives
- ◆ CFLAGS
 - Paramètres à passer au compilateur C
- ◆ LDFLAGS
 - Paramètres à passer à l'éditeur de lien
- ◆ EXEC
 - Nom de l'exécutable principal à générer

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDLAGS =
EXEC = hello

all: $(EXEC)
$(EXEC): hello.o main.o
    $(CC) -o $(EXEC) hello.o main.o $(LDFLAGS)
hello.o: hello.c hello.h
    $(CC) -o hello.o -c hello.c $(CFLAGS)
main.o: main.c hello.h
    $(CC) -o main.o -c main.c $(CFLAGS)

clean:
    rm -f *.o
mrproper: clean
    rm -f *~ $(EXEC)
```

Variables automatiques

- ◆ \$@
 - Nom de la cible de la règle
- ◆ \$<
 - Nom de la première dépendance
- ◆ \$?
 - Nom de toutes les dépendances qui sont plus récentes que la cible
- ◆ \$^
 - Nom de toutes les dépendances d'une cible
- ◆ \$*
 - Radical associé à une règle implicite
 - Si la cible est `toto.o` désignée par `%.o`, alors `$*` est `toto`

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS = -Wall -ansi
EXEC = hello

all: $(EXEC)
$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)
hello.o: hello.c hello.h
    $(CC) -o $@ -c $< $(CFLAGS)
main.o: main.c hello.h
    $(CC) -o $@ -c $< $(CFLAGS)

clean:
    rm -f *.o
mrproper: clean
    rm -f *~ $(EXEC)
```

Règles implicites

◆ Objectif

- ❑ Construction d'un module objet (.o) à partir d'un fichier source (.c)
- ❑ Règles appelées automatiquement pour construire toutes les cibles ayant pour suffixe .o

◆ Syntaxe

```
% .o: % .c  
commandes
```

◆ Notation plus ancienne

```
.c.o:  
commandes
```

Application

```
CC = gcc  
CFLAGS = -Wall -ansi  
LDFLAGS = -Wall -ansi  
EXEC = hello  
  
all: $(EXEC)  
$(EXEC): hello.o main.o  
$(CC) -o $@ $^ $(LDFLAGS)  
%.o: %.c  
$(CC) -o $@ -c $< $(CFLAGS)  
  
clean:  
rm -f *.o  
mrproper: clean  
rm -f *~ $(EXEC)
```

Problème de dépendance

- ◆ Dans l'exemple précédent
 - `main.o` n'est plus reconstruit si `hello.h` est modifié
- ◆ Solution
 - Préciser les dépendances séparément des règles implicites

◆ Illustration

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

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS = -Wall -ansi
EXEC = hello

all: $(EXEC)
$(EXEC): hello.o main.o
$(CC) -o $@ $^ $(LDFLAGS)
main.o: hello.h
hello.o: hello.h
%.o: %.c
$(CC) -o $@ -c $< $(CFLAGS)

clean:
rm -f *.o
mrproper: clean
rm -f *~ $(EXEC)
```

Problèmes avec les règles utilisateur

◆ Contexte

- ❑ `clean` est la cible d'une règle ne présentant aucune dépendance
- ❑ Supposons que `clean` soit également le nom d'un fichier présent dans le répertoire courant

◆ Problème

- ❑ Le fichier sera toujours plus récent que ses dépendances
- ❑ La règle ne sera jamais exécutée

◆ Solution

- ❑ Cible particulière → `.PHONY`
 - ❑ Dépendances systématiquement reconstruites
- ❑ `.bashrc` `PATH=/opt/jumble/bin:$PATH`
 - Ou `alias make='/opt/jumble/bin/make'`

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS = -Wall -ansi
EXEC = hello

all: $(EXEC)
$(EXEC): hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)
main.o: main.c hello.h
hello.o: hello.c hello.h
%.o: %.c
    $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper
clean:
    rm -f *.o
mrproper: clean
    rm -f *~ $(EXEC)
```

Substitution de variable

◆ Principe

- Définition d'une variable par substitution des éléments d'une autre variable

◆ Syntaxe

```
$(VARIABLE:OLD=NEW)
```

◆ Utilisation

- Construction de la liste des modules objets à partir des fichiers sources
- SRC
 - » Variable contenant la liste des fichiers sources du projet
- OBJ
 - » Variable contenant la liste des fichiers objets du projet

```
OBJ= $(SRC:.c=.o)
```

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS =
EXEC = hello
SRC = hello.c main.c
OBJ = $(SRC:.c=.o)

all: $(EXEC)
$(EXEC): $(OBJ)
    $(CC) -o $@ $^ $(LDFLAGS)
main.o: hello.h
hello.o: hello.h
%.o: %.c
    $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper
clean:
    rm -f $(OBJ)
mrproper: clean
    rm -f *~ $(EXEC)
```

Gestion des fichiers sources

◆ Objectifs

- ❑ Générer automatiquement la liste des fichiers sources
- ❑ Attention
 - » À utiliser avec précautions

◆ Méthode

- ❑ Utilisation de joker à la shell (`*.c`) pour obtenir tous les fichiers d'extensions `c` du répertoire courant
- ❑ Utilisation des caractères joker par la commande `wildcard`

◆ Exemple

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

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS =
EXEC = hello
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)

all: $(EXEC)
$(EXEC): $(OBJ)
    $(CC) -o $@ $^ $(LDFLAGS)
main.o: hello.h
hello.o: hello.h
%.o: %.c
    $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper
clean:
    rm -f *.o
mrproper: clean
```

Makefile conditionnel

- ◆ Exécution conditionnelle d'une partie d'un makefile
- ◆ Principe proche des `#ifdef` en C
- ◆ Utilisation
 - Définir des valeurs de variables différentes en fonction du contexte
- ◆ Exemple

```
ifeq ($(DEBUG),yes)
    CFLAGS=-Wall -ansi -g
    LDFLAGS=-g
else
    CFLAGS=-Wall -ansi
    LDFLAGS=
endif
```

Application

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS =
EXEC = hello
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
DEBUG = yes

ifeq ($(DEBUG),yes)
    CFLAGS=-Wall -ansi -g
    LDFLAGS=-g
    echo "Generating debug mode"
else
    CFLAGS=-Wall -ansi
    LDFLAGS=
    echo "Generating release mode"
endif
```

```
all: $(EXEC)

$(EXEC): $(OBJ)
    $(CC) -o $@ $^ $(LDFLAGS)
main.o: hello.h
%.o: %.c
    $(CC) -o $@ -c $<
    $(CFLAGS)
.PHONY: clean mrproper
clean:
    rm -f *.o
mrproper: clean
    rm -f *~ $(EXEC)
```

Génération automatique des dépendances

```
CC = gcc
CFLAGS = -Wall -ansi
LDFLAGS =
EXEC = hello
SRC = hello.c main.c
OBJ = $(SRC:.c=.o)
DEPFLAGS =

all: dep $(EXEC)
dep: Makefile.dep
Makefile.dep: $(SRC)
    @touch Makefile.dep
    $(CC) -MM $(DEPFLAGS) $(SRC) > $@
$(EXEC): $(OBJ)
    $(CC) -o $@ $^ $(LDFLAGS)
%.o: %.c
    $(CC) -o $@ -c $< $(CFLAGS)

.PHONY: clean mrproper dep
clean:
    rm -f $(OBJ)

mrproper: clean
    rm -f *~ $(EXEC) Makefile.dep

ifneq ($(wildcard Makefile.dep),)
    include Makefile.dep
endif
```

Techniques de programmation avancée

33

Conventions de nommage

- ◆ 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 bibliothèques (\$(exec_prefix)/lib)
- ◆ datadir
 - Répertoire d'installation des données statiques pour le programme (\$(exec_prefix)/lib)
- ◆ statedir
 - Répertoire d'installation des données modifiables par le programme (\$(prefix)/lib)

Techniques de programmation avancée

34

Conventions de nommage (suite)

- ◆ `includedir`
 - Répertoire d'installation des en-têtes (`$(prefix)/include`)
- ◆ `mandir`
 - Répertoire d'installation des fichiers de manuel (`$(prefix)/man`)
- ◆ `manxdir`
 - Répertoire d'installation des fichiers de la section x du manuel (`$(prefix)/manx`)
- ◆ `infodir`
 - Répertoire d'installation des fichiers info (`$(prefix)/info`)
- ◆ `srcdir`
 - Répertoire d'installation des fichiers source (`$(prefix)/src`)