UNIVERSITÉ
CÔTE D'AZUR

make & Makefile

Présentation: **Stéphane Lavirotte**
Auteurs: ... et al*

(*) Cours réalisé grâce aux documents de :
Erick Gallesio, Stéphane Lavirotte

Mail: Stephane.Lavirotte@univ-cotedazur.fr
Web: http://stephane.lavirotte.com/
Université Côte d'Azur

UNIVERSITÉ CÔTE D'AZUR

Principes général

- ✓ `make` (commande)
  - Construit automatiquement des fichiers
  - Exécute des commandes uniquement si elles sont nécessaires
  - Arriver à un résultat en ne faisant que les étapes nécessaires
    - Compilation/Installation de logiciel (exécutable ou bibliothèque)
    - Génération/Installation de documentation/pages html
    - …
- ✓ `Makefile` (fichier)
  - explicites les dépendances sous forme de règles

```
cible [cible …]: [dépendance …]
→ commande 1
→ …
→ commande n
```

→ = Tabulation

UNIVERSITÉ CÔTE D'AZUR

Principes dans le cas de la compilation

- ✓ **Une commande:** `make`
  - simplifie le processus de compilation
  - assure que les composants d'un projet sont dans un état cohérent
  - permet d'éviter les compilations inutiles
- ✓ **Un fichier de spécifications:** `Makefile`
  - explicites les dépendances d'un projet

```
/* Fichier main.c */

extern void say_hello(void);
Int main() {
    say_hello();
}
```

```
/* Fichier hello.c */

#include <stdio.h>
void say_hello(void) {
   printf("Hello, world!\n");
}
```

UNIVERSITÉ CÔTE D'AZUR

Makefile

```
# Makefile du programme précédent
main: main.o hello.o
	gcc -o main main.o hello.o
main.o: main.c hello.h
	gcc -c main.c
hello.o: hello.c hello.h
	gcc -c hello.c
```

Usage

```
$ make
gcc -c main.c
gcc -c hello.c
gcc -o main main.o hello.o
$ gedit main.c
.....
$ make
gcc -c main.c
gcc -o main main.o hello.o
```

UNIVERSITÉ CÔTE D'AZUR

Macros

- ✓ `make` **permet de définir des macros**
  - **définition avec** `macro=valeur`
  - `valeur` **peut être lue avec** `$(macro)`

```
OBJ = main.o hello.o
CC = gcc
CFLAGS = -DDEBUG –g –Wall –std=c99
main: $(OBJ)
    $(CC) -o main $(OBJ)
main.o: main.c hello.h
    $(CC) $(CFLAGS) -c main.c
hello.o: hello.c hello.h
    $(CC) $(CFLAGS) -c hello.c
```

- ✓ **Usage**

```
$ touch hello.c; make
gcc -DDEBUG -g –Wall –std=c99 helloc.c
cc -o main main.o hello.o
```

UNIVERSITÉ CÔTE D'AZUR

Macros: Types d'Affectation

- ✓ Il existe plusieurs manières de définir une macro :
  - = affectation par référence
    - expansion récursive
  - := affectation par valeur
    - expansion simple
  - ?= affectation conditionnelle
    - n'affecte la macro que si cette dernière n'est pas encore affectée
  - += affectation par concaténation
    - suppose que la macro existe déjà

UNIVERSITÉ CÔTE D'AZUR

Cibles fictives

- **Les cibles d'un `makefile`**
  - **fichier (la plupart du temps)**
  - **nom fictif (on dénote une action dans ce cas)**

```
# Ajouter ces lignes au Makefile précédent
clean:
    /bin/rm -f main $(OBJ)
print:
    lpr main.c hello.c
```

- **Usage:**

```
$ make clean
/bin/rm -f main main.o hello.o
$ make print
lpr main.c hello.c
```

UNIVERSITÉ CÔTE D'AZUR

Macro spéciales

- ✓ $@ *nom de la cible*
- ✓ $< *nom de la première dépendance*
- ✓ $^ *liste des dépendances*
- ✓ $? *liste des dépendances plus récentes que la cible*
- ✓ $* *nom du fichier sans suffixe*

```
# Makefile (nouvelle version)
....
main.o: main.c hello.h
    $(CC) $(CFLAGS) -c $*.c
hello.o: hello.c hello.h
    $(CC) $(CFLAGS) -c $<

print: .print

.print: hello.c main.c
    lpr $?
    touch .print
```

UNIVERSITÉ CÔTE D'AZUR

Règles Implicites

- ✓ `make` **connaît un certain nombre de règles implicites**
  - **nom du compilateur C rangé dans** `CC`
  - **option de compilations rangées dans** `CFLAGS`
  - **passage de** `.c` → `.o`:

```
$(CC) $(CFLAGS) -c fich.c
```

- ✓ **Simplifications du** `Makefile` **précédent:**

```
main.o : main.c hello.h
hello.o: hello.c hello.h
```

ou encore

```
main.o : hello.h
hello.o: hello.h
```

ou encore

```
$(OBJ): hello.h
```

- ✓ **Note: Les dépendances peuvent être calculées par** `gcc` **(options** `-M` **ou** `-MM`**)**

UNIVERSITÉ CÔTE D'AZUR

Définition de Règle Implicites

- ✓ Les règles implicites peuvent être définies avec .SUFFIXES

```
.SUFFIXES: .c .c.gz
.c.c.gz:
    gzip -9 $<
.c.gz.c:
    gunzip $<
```

- ✓ Usage

```
$ make hello.c.gz main.c.gz
gzip -9 hello.c
gzip -9 main.c
```

→

```
clean: hello.c.gz main.c.gz
/bin/rm -f main $(OBJ)
```

UNIVERSITÉ CÔTE D'AZUR

GNU make goodies

- La commande `make` de GNU permet de définir autrement les règles implicites

```
%.c.gz: %.c
    gzip -9 $<
%.c: %.c.gz
    gunzip -9 $<
```

- Autres commandes GNU utiles (voir la doc pour une liste complète)

```
$(wildcard $.c)
$(VAR:.c=.exe)
```

- Voir un exemple d'utilisation dans le transparent suivant

UNIVERSITÉ CÔTE D'AZUR

Le `Makefile` complet

- ✓ ***Un `makefile` réaliste pour notre hello world pourrait être:***

```
CC=gcc
CFLAGS=-Wall -std=gnu99 -O3

SRC=$(wildcard *.c)
OBJ=$(SRC:.c=.o)
EXE=helloworld

$(EXE): $(OBJ)
     $(CC) -o $(EXE) $(OBJ)

$(OBJ): hello.h

clean:
     rm -f $(OBJ) $(EXE) *~
```

- ✓ ***Exemple d'utilisation:***

```
$ make
gcc -Wall -std=gnu99 -O3 -c -o hello.o hello.c
gcc -Wall -std=gnu99 -O3 -c -o main.o main.c
gcc -o helloworld hello.o main.o
$ touch hello.c; make
gcc -Wall -std=c99 -O3 -c -o hello.o hello.c
gcc -o helloworld hello.o main.o
$ make clean
rm -f hello.o main.o helloworld *~
```

UNIVERSITÉ CÔTE D'AZUR

Un Makefile pour les TDs

```
CC=gcc
CFLAGS=-Wall -std=gnu99 –g

SRC=$(wildcard *.c)
EXE=$(SRC:.c=)

all: $(EXE)

clean:
rm -f $(EXE) *~
```

- ✓ ***Exemple d'utilisation:***

```
$ ls
Makefile exo1.c exo2.c
$ make
gcc -Wall -std=gnu99 -g exo1.c -o exo1
gcc -Wall -std=gnu99 -g exo2.c -o exo2
$ ls
Makefile exo1 exo1.c exo2 exo2.c
$ make clean
rm -f exo1 exo2 * ~
$ ls
Makefile exo1.c exo2.c
```

UNIVERSITÉ CÔTE D'AZUR

Conclusion

- ✓ Un outil ancien (1977), très puissant
  - « C'est dans les vieux pot que l'on fait les meilleures soupes »
  - Repose sur un double mécanisme simple: fichier et date
- ✓ Quelques limitations (pour être totalement honnête)
  - Rien n'assure que la date soit la date de dernière modification (ex pas sur un support réinscriptible
  - Tout changement de date (qui peut ne pas être du à un changement de contenu comme avec `touch`) a un impact
    - Si c'est une source: perte d'efficacité (regénère la cible)
    - Si c'est une cible: potentielle perte de cohérence/fiabilité (ex: on a fait une modification sur la source puis une modification de date sur la cible => on ne recompile pas)
  - Ignore la sémantique des fichiers
    - Avantage: peut ne pas être utilisé que pour compiler du C
    - Inconvénient: recompile même si on a ajouté un commentaire
- ✓ Les mécanismes plus récents spécifiques aux langages