UNIVERSITÉ DE NANTES

POLYTECH NANTES

Introduction à Makefile

Support de Cours
Dpt Informatique Polytech'Nantes
Fabien Picarougne

Automatisation de la compilation

- Compilation d'un programme contenant plusieurs sources

```
gcc -o calcul calcul.c input.c matrix.c graph.c -lm
```

- Problème
  - Commande longue à taper
    - Pour un programme classique !
  - Si un seul fichier est modifié, compilation de l'ensemble

- Solution
  - Compiler les éléments de manière séparée

```
gcc -c calcul.c
gcc -c input.c
gcc -c matrix.c
gcc -c graph.c
gcc -o calcul calcul.o input.o matrix.o graph.o -lm
```

  - De nombreuses commandes à taper : fastidieux !

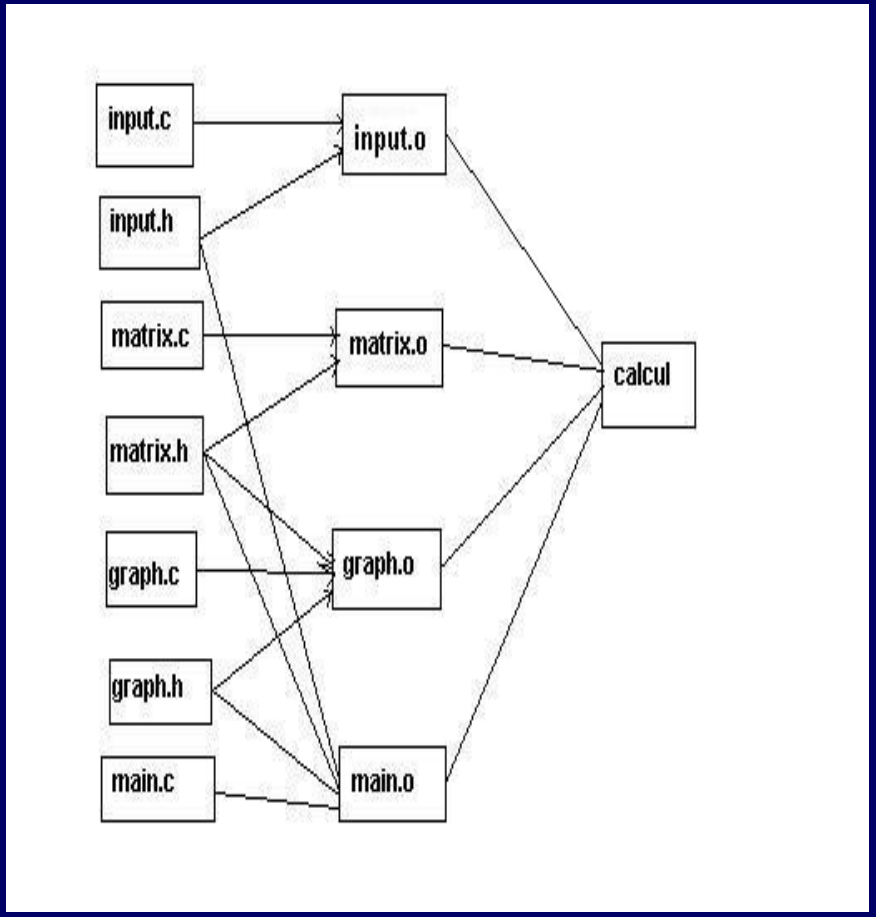
Automatisation de la compilation

- La commande `make` permet d'automatiser des actions
  - Archivage de document
  - Mise à jour de site
  - Compilation d'un projet

- Ordres donnés dans un fichier
  - Généralement *Makefile*
  - Sous forme de règles
    - cible : dépendance
      actions

Un 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



Un exemple

- Le fichier `Makefile` sert à définir les cibles, les dépendances et les commandes pour reconstruire les cibles

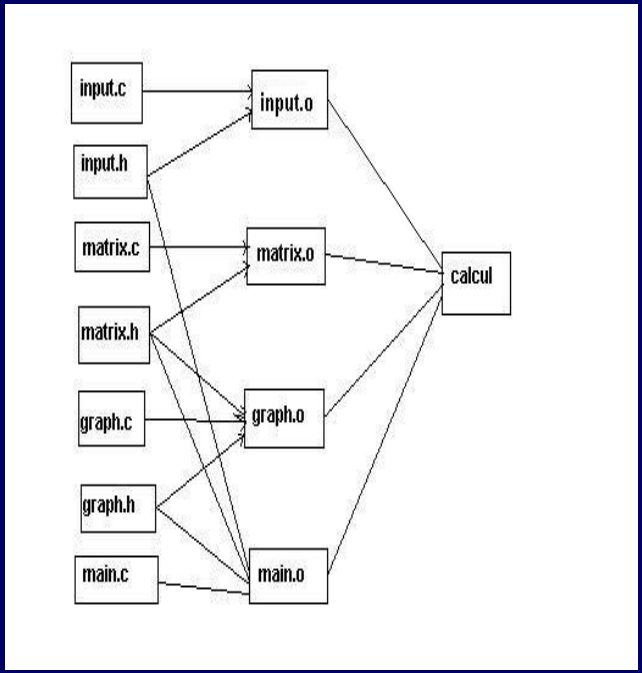
```
calcul: main.o input.o matrix.o graph.o
    cc -o calcul calcul.o input.o matrix.o graph.o -lm

main.o: main.c input.h matrix.h graph.h
    cc -c main.c

input.o: input.c input.h
    cc -c input .c

matrix.o: matrix.c matrix.h
    cc -c matrix.c

graph.o: graph.c graph.h matrix.h
    cc -c graph.c
```



Eléments d'un Makefile

- Commentaires introduits par `#`
- Règles explicites : comment refaire plusieurs fichiers
  - Cibles + dépendances : fichier à construire
  - Commandes pour produire la cible
- Règles implicites : règles génériques
  - Evite de lister tous les fichiers
  - Simplifie l'écriture d'un Makefile
- Variables
  - Permet de définir des listes de fichiers ou des options dans les commandes
- Directives
  - Permet de piloter le Makefile grâce aux arguments passés à `make`

Règles de construction

- Modèle

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

- Cibles et dépendances se trouvent sur la même ligne
- Le caractère « \ » peut forcer `make` à ignorer le retour à la ligne
- Les commandes commencent toujours par une tabulation et doit tenir sur une seule ligne avec éventuellement un \
- La liste de commandes se terminent par une ligne sans tabulation
- Un mot commençant par le caractère $ est utilisé pour désigner le contenu d'une variable

Commandes

- `make` n'affiche pas une commande qui commence par `@`
- 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
- `make` abandonne l'exécution d'une règle et de tout le processus en cas d'erreur
  - 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

Commandes : exemple

- Attention à mettre les commandes sur une ligne

```
dir/prog: a.o b.o
	cd dir
	cc –o prog ../a.o ../b.o
```

  - C'est incorrect, chaque ligne est un nouveau contexte shell

```
dir/prog: a.o b.o
	cd dir ;\
	cc –o prog ../a.o ../b.o
```

- Règles de nettoyage

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

- Cible particulières

```
.PRECIOUS
```

  - Indique les cibles qui ne seront pas détruites en cas d'interruption du make

Utilisation des variables

- Utilisé pour décrire des listes de fichiers, de programmes, d'options de commandes
- Nom : séquence de caractères sauf : `, # = " <tab> espace`
- Référence `$(nom de la variable)`

```
OBJECTS= a.o b.o c.o
prog:$(OBJECTS)
        cc -o prog $(OBJECTS)

$(OBJECTS) : common.h
```

- Il existe des variables automatiques définies par `make`
- `make OBJECTS=`
  - change la valeur de la variable `OBJECTS`

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

Règles implicites

- Deux types de règles implicites
  - Les règles incorporées à `make` mais qui dépendent de la version de `make`
  - Règles implicites définies par l'utilisateur

- Définition
  - Utilisation du caractère `%`

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

  - Utilisation des suffixes

```
.SUFFIXES: .o .c    # début de makefile
…
.c.o:
        $(CC) -c $(CCFLAGS) $(DEBUG) $(OPTLEVEL) $<
```

Cibles standard

- `all` : première règle explicite pour définir ce qui est fait par défaut
- `install` : installation du programme construit
- `uninstall` : défait ce que install a fait
- `clean` : supprime du répertoire courant tous les fichiers construits par `make all`
- `distclean` : remet les répertoires sources dans leur état initial
- Une seule cible par répertoire facilite l'écriture et la maintenance de logiciels

Substitutions de variables

- Syntaxe

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

- Exemple1

```
SRC=toto.c
OBJ=$(SRC:.c=.o)   -> OBJ= toto.o
```

- Exemple2

```
SRCS=toto.c titi.c
OBJS=$(SRCS:%.c=%.o)   -> OBJS= toto.o titi.o
```

- Exemple3

```
BIB=bib.a
SRCS=a.f b.f c.f
ELTS=$(SRCS:%.c=$(BIB)(%.o))
    -> ELTS = bib.a(a.o) bib.a(b.o) bib.a(c.o)
```

Appel récursif de make

```
SUBDIRS= a b c d
all :
      @for i in $(SUBDIRS) ;\
      do;\
        (cd $$i;\
        echo "Faire tout dans $$i";\
        $(MAKE) all \
        ) || exit 2;\
      done
```

- `@` : pour supprimer l'affichage
- `\` : maintenir la commande SHELL sur la même ligne
- `$$` : insérer le $ dans la ligne du SHELL
- `;` : séparer les instructions dans un même SHELL
- `exit` : force une erreur dans le shell

Production automatique de dépendances

- `make` devient inutile si les dépendances ne sont pas à jour
- Problème de génération automatique de dépendance
- En C /C++:
  - Gestion des dépendances entre les classes, les fichiers entêtes, les objets…
  - Utilitaire `makedepend`
  - Options de compilation de `gcc`

Appel du make

- `make` utilise par défaut le fichier Makefile ou makefile
- `make -f <fich>` permet de spécifier un fichier
- `make` hérite des variables d'environnement
- L'option `-k` impose à `make` de continuer lorsqu'il rencontre une erreur
- L'option `-s` supprime les affichages
- L'option `-n` supprime l'exécution et affiche les commandes