INF203 - Makefiles

Victor Lambret

2017

Le besoin

La compilation séparée

- Avec la compilation séparée on ne recompile pas tout le projet après une modification
- Idéalement, on voudrait même ne recompiler que ce qui est nécessaire !
- Recompiler tout à la main c'est facile.
- Recompiler ce qui est nécessaire à la main c'est compliqué.
- Facilitons nous la vie !

Quand recompiler ?

- Quand on modifie le code source il faut recompiler le programme
- Il faut des règles pour savoir ce qu'il faut recompiler !
- Ces règles dépendent du type de fichier

Modification d'un fichier.c

- Quand on modifie un fichier.c on doit recompiler le fichier.o associé
- Comme on modifie le fichier.o on doit recompiler le programme complet qui est compilé à partir des fichiers .o

Modification d'un fichier.h

- Quand on modifie un fichier.h on modifie en général l'interface publique de fonctions
- On veut recompiler le fichier.c associé pour vérifier que les fonctions correspondent à la nouvelle interface
- On veut recompiler les fichiers .c incluant le .h afin de vérifier qu'ils utilisent correctement la nouvelle interface
- Comme on recompile les fichiers .c on crée de nouveaux .o donc recompilation du programme

Observations

- On compile pour créer un exécutable
- On utilise des fichiers intermédiaires
- Chaque fichier intermédiaire est créé à partir d'un ou plusieurs fichiers sources

Structure en graphe

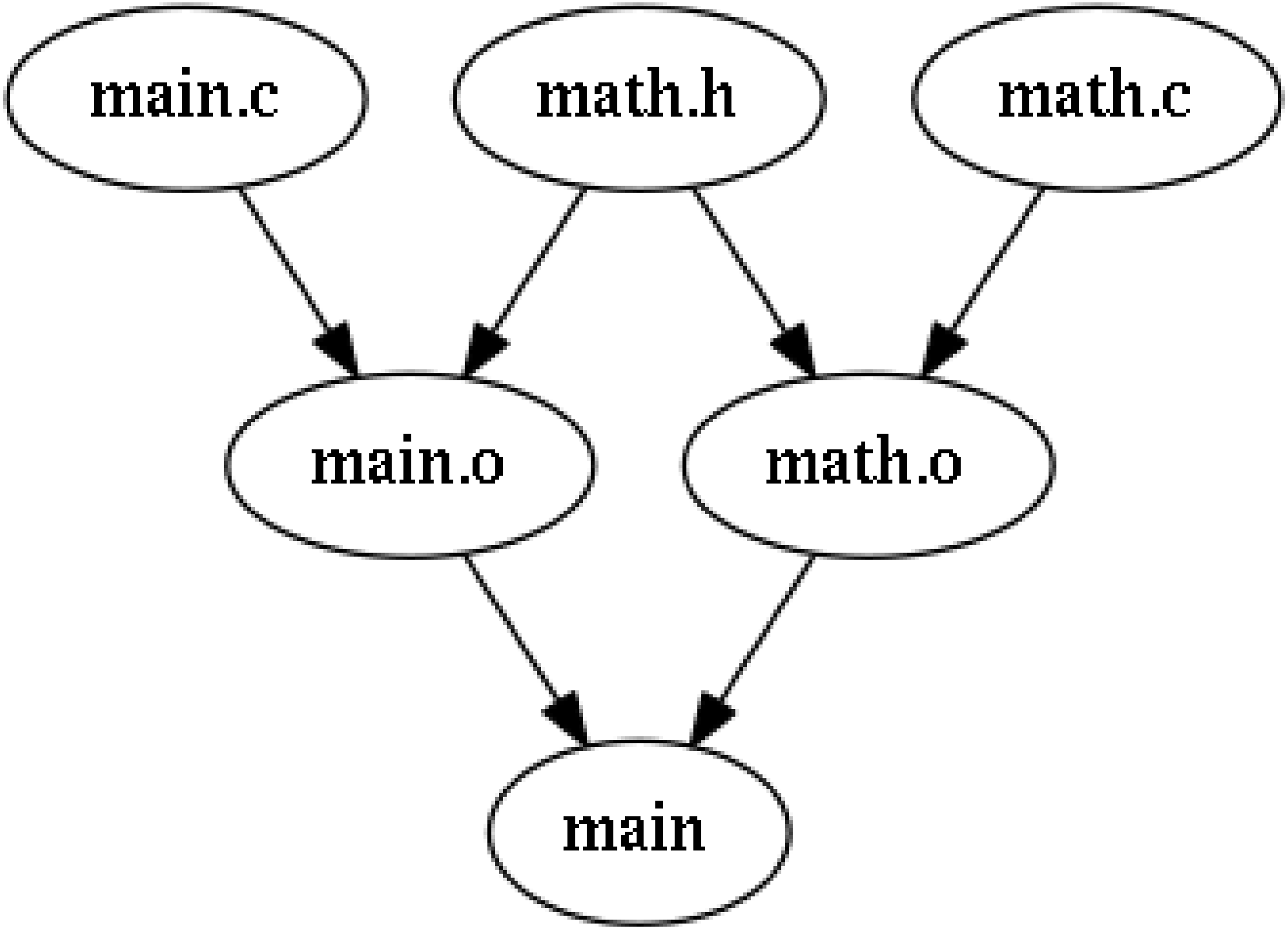
- On peut créer un graphe orienté modélisant ce genre de compilation
- Une racine : l'exécutable
- Des feuilles : le code source
- Le reste : des fichiers intermédiaires

Cas d'étude : code source

Nous allons prendre pour la suite un code assez simple :

```
main.c
math.c
math.h
```

Cas d'étude : graphe



Automatiser le build

Afin d'automatiser le build il faut :

- Une description de ce graphe
- Chaque transition représente une dépendance
- On doit indiquer une commande pour construire chaque fichier intermédiaire ou final

Plusieurs approches sont possibles, l'une d'elle est d'utiliser la commande make !

Make

Historique

- Ce besoin de build automatique existe depuis longtemps
- Au début d'Unix il existait des scripts dédiés pour chaque application
- Première version de l'outil make tel qu'on l'utilise : 1977

Le Makefile

- la commande make repose sur un fichier Makefile décrivant le projet
- Ce Makefile décrit les dépendances et les commandes pour construire le système
- `make` recherche le Makefile dans le répertoire courant

Syntaxe générale :

Un makefile contient un ensemble de règles de ce genre :

```
cible : dependance1 dependance2
    commandes pour
    construire la cible
```

Important : erreur classique !

- Pour des raisons historiques les commandes d'un Makefile doivent être indentées avec une tabulation.
- Une erreur classique est d'utiliser un espace, on obtient alors l'erreur suivante :

```
Makefile:8: *** missing separator (did you mean TAB
instead of 8 spaces?). Arrêt.
```

Pensez à bien configurer votre éditeur de texte pour indenter avec des tabulations !

Exemple de Makefile

```
main : main.o math.o
    gcc main.o math.o -o main

main.o : main.c math.h
    gcc -c -Wall -Werror -o main.o main.c

math.o : math.c math.h
    gcc -c -Wall -Werror -o math.o math.c
```

Utilisation

- Pour compiler le programme on peut taper : `make target`
- `target` peut-être n'importe quelle cible décrite dans le Makefile
- `make` va recompiler ce qui est nécessaire pour recompiler la cible
- Si rien n'est nécessaire, `make` n'exécute aucune commande
- `make` est pensé pour que les cibles soient des noms de fichiers

Quand recompiler une cible ?

- Si le fichier de la cible n'existe pas
- Si une de ses dépendances doit être recompilée
- Si le fichier d'une de ses dépendances est plus récent que la cible

Déclaration de variable

On peut déclarer une variable ainsi :

```
VAR=valeur
VAR = valeur
```

A l'inverse du bash on peut utiliser des espaces autour du signe =

Utilisation de variable

Pour utiliser une variable, on doit l'utiliser de la manière suivante :

`$(VAR)`

- A l'inverse du bash, on **doit** utiliser des parenthèses autour du nom de variable
- Tout comme le bash une variable non définie ne cause pas d'erreur et contient une chaîne vide

Exemple de Makefile avec variables

On peut utiliser les variables pour factoriser des répétitions :

```
CC=gcc
CLAFGS=-Wall -Werror

main : main.o math.o
    $(CC) main.o math.o -o main

main.o : main.c math.h
    $(CC) -c $(CFLAGS) -o main.o main.c

math.o : math.c math.h
    $(CC) -c $(CFLAGS) -o math.o math.c
```

La première règle

- Un appel à `make` sans argument construit la première cible du Makefile
- Un usage courant est de créer une première règle all qui indique ce qu'il faut construire :

```
all : main
```

Aller plus loin avec make

- `make` est un outil très puissant et générique
- On peut l'utiliser pour autre chose que de la compilation : par exemple pour générer ces slides !
- Il existe plein de fonctionnalités pour se simplifier la vie, ce qu'on voit ici est le fonctionnement de base

Autres systèmes de build

- `make` est un outil ancien qui comporte des défauts mais fait bien son travail
- Il existe de nombreux remplaçants pour gérer des compilations automatiques (cmake, ninja, etc.)
- Le modèle de fonctionnement de `make` reste une référence : de nombreux outils de remplacement génèrent des fichiers de Makefile